POPULAR APPLICATIONS SERIES

# WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++

Learn how to create your own applications language that will work with your C++ or Turbo/Borland C++ programs. With this knowledge, you will be able to provide a useful set of commands to users of your programs to increase their speed, flexibility, and ease of use. Underlying theory and practical examples make this book a must for every programmer. Source diskette included.

NORMAN E. SMITH is a senior systems analyst and programmer for Science Applications International, Corp. with over 20 years of experience with computer programming languages and applications. Mr. Smith is also an instructor and lecturer on several applications, as well as numerous programming languages including C, C++, FORTH, FORTRAN, COBOL, and BASIC.

**Book Buyer's Guide**

User Category: ○ New ● Experienced ● Advanced ● Professional ● Educator

Book Type: ○ Tutorial — for: ● Self Teaching ○ Classroom Use
○ Command Reference
● Advanced Topics
● Working Examples — ● Source Code ○ Templates ○ Hands-on Activities ○ Other
● Companion Diskette — ● Included ○ Available

9 781556 222641

ISBN 1-55622-264-5 $14.95

Cover Design and Photo by Alan McCuller

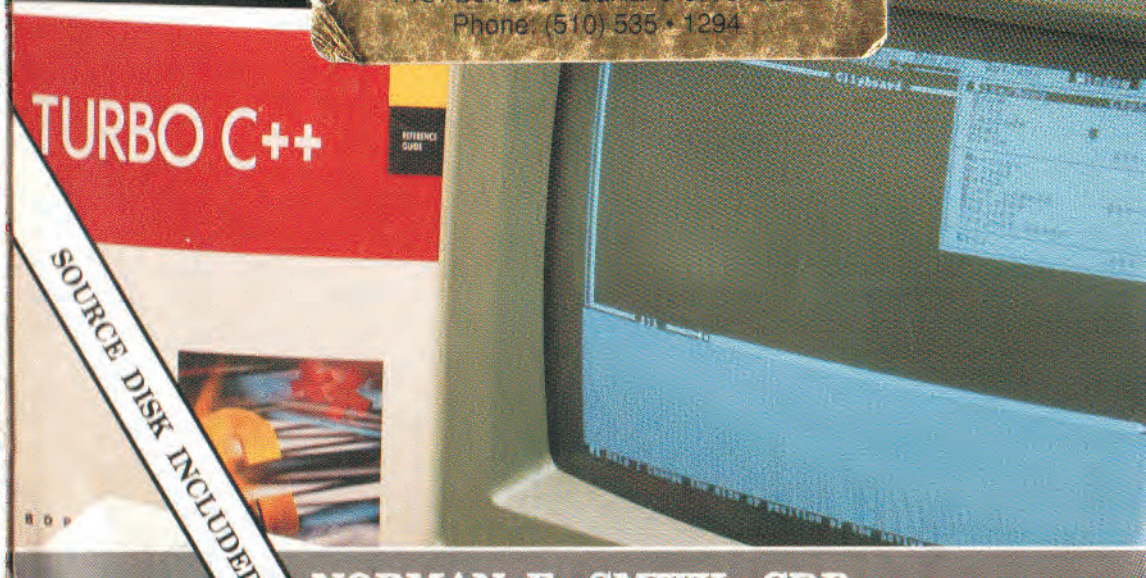
WORDWARE PUBLISHING, INC.
—First in Quality—

POPULAR APPLICATIONS SERIES
WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++™
SMITH

POPULAR APPLICATIONS SERIES

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++™





# Write Your Own Programming Language Using C++

Norman E. Smith, CDP

Wordware Publishing, Inc.

**Library of Congress Cataloging-in-Publication Data**

Smith, Norman E. (Norman Earl), 1952-
Write your own programming language using C++ / Norman Earl Smith.
p. cm.
Includes index.
ISBN 1-55622-264-5
1. C++ (Computer program language). 2. Application software.
I. Title.
QA76.73.C153S66 1991
005.13--dc20 91-44231
CIP



1506 Capital Avenue
Plano, Texas 75074



Printed in the United States of America

ISBN1-55622-264-5

10 9 8 7 6 5 4 3 2
9111



All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(214) 423-0090

# Dedication

For My Wife Sandi,

For putting up with my writing two books!

# Contents

# Preface

This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation.

An application language is a language embedded in an application that provides the user the capability to write small programs, often called macros, to automate repetitive tasks or otherwise extend the usefulness of the application. Many commercial programs include application languages, even though it may be called something else. Lotus 1-2-3 has a macro language. WordStar has named macros. dBASE has a language. AutoCAD has AutoLisp. The list goes on and on. *Write Your Own Programming Language Using C++* provides the ordinary C++ programmer the tools necessary to build an application language into any application program.

# Chapter 1

# ABOUT THIS BOOK

## INTRODUCTION

Writing a custom application language is much easier than you may believe. It's a matter of knowing where to get started. Many commercial products include application languages. Autocad has AutoLisp, and Procomm Plus includes ASPECT. An application language adds a professional quality to any program.

This book covers implementing a simple application language, CALC, that can be added to any interactive C++ program written with either Borland C++ or Turbo C++, which are popular products of Borland International, Inc. CALC is implemented using a threaded interpreter called Until for UNconventional Threaded Interpretive Language and an RPN compiler named CALL for Callable Application Language Library.

The first chapters cover CALC and a sample program that uses it. The second group of chapters present the underlying software technology (the Until interpreter and CALL compiler) for implementing application languages.

The code presented in this book was developed under Borland C++.

This book assumes that you have Borland C++ or Turbo C++ installed and operating on your PC and are familiar with editing and compiling programs in that environment. Care was taken to keep the code as generic as possible, so converting it to run with another C++ or even C compiler should not be difficult.

## WHY WRITE A CUSTOM APPLICATION LANGUAGE

There are many reasons for writing a custom application language. An applications language is a good way to add functionality a user may need that is not directly related to the program. A good example is a calculator language that allows the user to write new macros.

An application language is useful anytime repetitive or complicated commands are required to accomplish an action.

This is a primary reason that virtually every commercial word processor, spreadsheet, and communications program has some sort of macro or application language. Capabilities range from simply recording and playing back keystrokes in PFS: Professional Write to a full programming language like Mocklisp in GNU Emacs.

Programs that include full languages provide the user with a means to extend or customize the application to fit their day-to-day environment. If a common core macro/application language is used across all applications, there is a consistency that users become familiar with. Training and documentation costs can be reduced via application consistency.

Users also tend to select products that they are comfortable with. A common application language can go a long way toward supplying consistency.

Application languages can allow user access to internal program data structures without having to provide source code or object libraries. For example, an application language can let the user modify the number of lines per page on a report.

Another major reason for writing an application language is program testing. Access to program data structures is very easy to include in an interactive application language. Manipulating compiled variable values can speed up testing considerably. This is not as important when using Borland C++ because of its built-in debugging environment. If your C/C++ compiler does not have these features, interactive access to variables can be a significant time saver when debugging a program.

## IMPLEMENTING AN APPLICATION LANGUAGE

There are any number of approaches to implementing a macro language or application language. These include:



- Record keystrokes in a buffer
- Predefine commands to execute
- Compile new commands to execute
- A threaded interpreter

Little knowledge of compilers is required to develop your own applications language provided you have the proper tools, such as those presented in this book. A sample application language is developed here, including the compiler and interpreter. The application language is named CALC, the compiler CALL, and the interpreter Until.

The following sections take a cursory look at possible implementation approaches for an applications language.

**RECORDED KEYSTROKES** The simplest approach to implementing a macro or applications language is recording keystrokes. Every keystroke is copied into a buffer or buffers and saved for later recall. Generally, a macro sequence is assigned to a particular keystroke, such as Alt-A, or assigned a name. Rarely are looping or conditional logic capabilities included. Word processors frequently follow this approach.

**PREDEFINED COMMANDS** Some programs take a slightly different approach to implementing application language type processing. The functions provided are completely defined by the developer. An example of the predefined command approach is illustrated by the automatic dialing facilities of many communications programs. They escape normal operation mode to allow the user to manipulate the modem to dial the telephone. Predefined commands are generally bound to a specific command key combination such as Alt-D.

**COMPILE COMMANDS** Some programs include simple compilers specific to the application. Examples are ASPECT in Procomm Plus and AutoLisp in AutoCAD. These may or may not be full compilers in the traditional sense. In any case, they are complex to implement.

**THREADED INTERPRETER** A Threaded Interpreter, or Threaded Interpretive Language (TIL), is a combination compiler and interpreter all rolled into one. Procedures, or functions, or words, or macros, depending on the TIL, are compiled. A separate interpreter executes the "threads" the compiler builds. TILs are generally interactive. The best known TIL is Forth. TILs can be quite a bit faster than interpretive Basic. TIL speeds approach the performance of typical compilers of the mid-eighties. The Microsoft Quick languages are reported to be based on a threaded technique.

## WRITING STYLE

Knowledge about the writing conventions followed in this book is helpful in reading the text.

- C++ identifiers and reserved words are set in italic type.
- Two-key combinations are a combination of a shift key and a letter.

The shift keys are Alt, Ctrl, and Shift. For example, "Press Ctrl-C to exit." means press the 'C' key while holding the Ctrl key down. The word "Press" indicates a key or key combination should be pressed to continue.

## ORGANIZATION

Each chapter covers a single topic. It consists of an introduction to the topic or feature. Examples are presented where appropriate. The introduction serves as a formal definition of the topic to be covered. A Chapter may include other sections necessary to fully explain the given topic.

## CODING STYLE

The coding style used in this book favors clarity in the C++ code used in the examples. The coding style of many C programmers combines statements where possible. The typical way of testing the length of a string for 9 is:

```
if(strlen(xxx)==9){ ... }
```

The same code following coding style in this book is:

```
len = strlen(xxx);
if(len == 9){
        ...
}
```

Since new concepts are being presented, the readability of the code is more important than absolute program efficiency.



# Chapter 2

# GUIDED TOUR THROUGH CALC

## INTRODUCTION

The previous chapter introduced the concept of the application language and an example, CALC. This chapter provides a guided tour through CALC. The tour consists of a sample session using CALC compiled as a stand-alone program. The examples in this chapter introduce CALC and attempt to show the potential of an application language as a useful part of your C++ toolbox.

CALC uses RPN-type operators, with arguments specified before the operator. CALC uses RPN rather than infix notation because that is how the compiler is written. CALC uses Until to provide interactive interpreter capabilities and CALL to compile macros. The fact that Until is a Threaded Interpretive Language does not imply application languages built on top of it must be RPN. Until is simply the interpreter CALC uses. The compiler determines whether the application language uses RPN or infix or postfix operator notation. C/C++ uses infix notation and Lisp uses postfix notation. The CALL compiler is discussed in detail in later chapters.

This chapter has a sample session to give a feeling of CALC's operation. The examples also touch on compiling new CALC macros. An overview of the pieces that make up the application language and a section that presents some uses for CALC and Until other than just an application language are also presented.

## RUNNING CALC AND THE BASICS

Copy the files CALC.EXE, CALC.APP, and HELP.APP from the floppy disk included with this book to your hard disk. The examples introduce you to many of the built-in CALC commands.

1. Type **CALC** from the DOS prompt, then press **Enter** to start a CALC session.

```
                    CALC
              Copyright 1992
          All Rights Reserved

                Version 1.1
              JANUARY 1992

CALC> Welcome to the CALC Programmable Calculator

Type 'macros' to show list of available macros.
Type 'help help' to type the macro usage.
Type 'exit' to exit CALC and return to DOS.
```

You may type **hello** at any time to type the startup message again. The upper right-hand corner of the screen displays the Data Stack most of the time. When the Data Stack is empty, the display is 'Stack Empty.' The display is updated each time a number is pushed onto the stack or a macro executed.

### NOTE

Type *exit* then press the Enter key at any time to exit CALC and return to DOS.

2. Type **1 2 + =** and press **Enter** to add 1 and 2. The = causes the result, 3, to print.

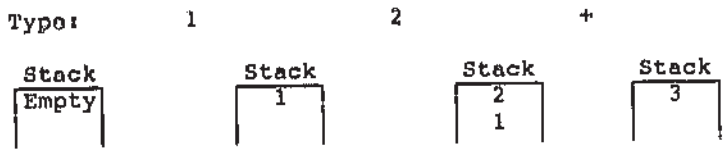
```
CALC> 1 2 + =
3
```

### NOTE

CALC operators are space delimited and case sensitive. That is, a space between operators is required, and commands CMD, Cmd, and cmd are all different.

CALC uses Reverse Polish Notation (RPN) like an HP Calculator to pass parameters. Typing a number pushes its value onto the top of the stack. The following illustrates each step executed by CALC in adding two numbers.






Execute the s? macro at any time to type the contents of the stack to the computer screen.

### NOTE

Parameters come first when using RPN notation. If you have a CALC macro named xxx that takes two parameters, you invoke xxx with the following command:

```
1 2 xxx
```

Assume you want to call the function xxx in C (infix notation) that requires two parameters. You would use:

```
xxx(1, 2);
```

to execute it.

3. Type **1 2 s?** and press **Enter**. The 1 and 2 are pushed onto the stack and s? causes the stack contents to print without modifying the stack:

```
CALC>1 2 s?
Stack
1
2
```

### NOTE

Press Enter to execute each command sequence specified in the examples.

4. Now type **=**, then press **Enter**. The number on the top of the stack is displayed by the = operator. 2 prints because it is the top value on the stack. Type = again to print the 1.

A list of the built-in CALC commands are included in Appendix C. All of the currently defined macros can be displayed by the *macros* command or examples of how to use each command displayed with the *help* command.

5. Type **macros** then press **Enter** now.

```
CALC> macros

      hello          )          (          =
      endif       else         if          {
         }m         m{       help       exit
    do_loop      do_do     .quote      quote
     branch      0bran        mod         s?
      abort          3          2          1
          0    macros       drop         <-
         ->          *          /        rot
        dup       swap          ?          +
          -          <          >         ==
         cr          =      const        var
        lit     create          i
```

The macro list contains the names of each macro presently defined, including both internal macros such as *if* or *endif* and macros you define. CALC also includes a built-in help facility that prints a definition for the built-in operators. For example,

6. Type **help var** to display the help entry for var.

```
CALC> help var
var             { --- }                    Compile time
                { --- addr.of.var }        Run time
                "var". At compile time, creates a
                variable:
                        var trash
                At run time, push the address of the
                variable onto the stack.
                        trash    { --- addr.of.trash }
```

7. Type **exit** and press **Enter** to exit CALC and return to DOS.

This session provided a quick introduction to the interactive operation of CALC. CALC is an application language that implements a simple four-function HP-like calculator.

## THE NEXT STEP: COMPILING MACROS

Now that we have examined the interactive portion of CALC, let's look at defining macros. A CALC macro begins with m{ and ends with }m. Any of the macros used in an interactive session can be used in a macro definition. This section shows how to add new macros to CALC. Complete programs can be written in CALC. Some simple mathematical formulas are developed as macro examples.

Type **CALC** at the DOS prompt to run CALC.



### NOTE

CALC compiles the contents of CALC.APP at startup. You can add your own macro definitions to that file.

**HELLO WORLD** The traditional first program for most C programmers is some form of "Hello World." Here it is in CALC:

```
m{ Hello_World
    " Hello Cruel World..."
    }m
```

Now type the macro definition at the keyboard. The space after "m{" and the '"' are significant. After it has been compiled, type **Hello_World** to execute it.

```
CALC> Hello_World
Hello Cruel World...
CALC>
```

### NOTE

All of the macros in the following examples are indented to allow comments and show code structure. The comments are not necessary. (Comments are the text between { and }.) The code can be typed all on one or two lines for convenience.

**PRINT 1 TO n** The second macro performs a simple loop, counting from 1 to n, where 'n' is the number on the top of the stack. Type the following macro at the CALC prompt:

```
m{ 1-to-n               { n --- }
    ( i                 { Get the loop index          }
      1 +               { Add 1 to it                 }
      =                 { Type the value              }
    )                   { Close the loop              }
    cr                  { Type a carriage return      }
    }m                  { End the macro               }
```

Note that loop indexes start at 0, so if printing starts at 1, then 1 must be added to the loop index. The loop construct, "( ... )", can only be used inside a macro definition. Type the following command to execute 1-to-n:

```
CALC> 10 1-to-n
1 2 3 4 5 6 7 8 9 10
```

Try writing a count-from-0 macro. Hint: Leave out the "1 +" line.

**ABSOLUTE VALUE** A useful macro is calculating the absolute value of a number. It also illustrates the *if ... else ... endif* construct. An easy algorithm for calculating absolute value is to subtract the number from 0 if it is negative. Otherwise, it is already the absolute value. Type the following code at the CALC> prompt:

```
m{ abs                  { n --- }
    dup 0 <             { Duplicate n and compare to 0  }
    if                  { n is negative                 }
      0 swap -          { 0 - n to reverse the sign     }
      =                 { Type the |n|                  }
    else                { n is positive                 }
      =                 { Type the |n|                  }
    endif               { Close the if                  }
    }m                  { End the macro                 }
```

Execute abs with a positive number:

```
CALC> 50 abs
50
```

Execute abs with a negative number:

```
CALC> -27 abs
27
```

The first line of the *abs* macro definition has a strange-looking comment that is very important. *{ n --- }* after the macro name is called the stack comment. It contains the calling sequence to execute the command. The stack comment for abs means that a number, n, is expected on the top of the stack before execution. "---" represents the operation being performed. *abs* leaves nothing on the stack when execution has completed. Without the stack comment, you must figure out the macro parameters by reading the code.

**FIBONACCI NUMBERS** A Fibonacci set is a set of numbers where the Ith value is the sum of the previous two values. It is commonly used as a program benchmark. The set is:

```
1 1 2 3 5 8 13 21 ...
```

A macro to compute Fibonacci numbers is not complex. The majority of the code is a loop with a little stack manipulation. Type the following code at the CALC> prompt:

```
m{ fibonacci            { --- }
    1 dup = dup = dup   { Print the initial 1's         }
    10 (                { Do the loop 10 times          }
        dup             { Make extra copy of curr value }
        rot             { Move third stack item to top  }
        +               { This value + previous         }
        dup =           { Print latest number           }
    )                   { End of loop                   }
    drop drop           { Clean up leftovers on stack   }
    }m                  { Close out the macro           }
```

(Leave out the comments. The macro definition can be typed all on a single line. The indentations are used here to show the structure of the example better.)

Type **fibonacci**, then press **Enter** to execute the macro:

```
CALC> fibonacci
1 1 2 3 5 8 13 21 34 55 89 144
```

Try modifying the definition to take the number of times to loop from the command prompt. So you type:

```
20 fibonacci
```

to calculate 20 Fibonacci numbers instead of 10.

**FACTORIAL** The final example is another traditional compiler benchmark. A factorial is the product of a positive integer, n, multiplied by the product of all positive integers less than n. For example, 3 factorial is 6 or 1 * 2 * 3. There are a number of methods for computing a factorial. Again, the emphasis here is on simple for CALC examples. Type the following code at the CALC> prompt:

```
m{ factorial            { n --- }
    1 swap              { Start with 1                  }
    (                   { Begin the loop                }
      i 1 +             { Add 1 to the loop index       }
      *                 { Do the multiply               }
    )                   { Go to the top of the loop     }
    =                   { Type the results              }
    }m                  { End the macro definition      }
```

At the CALC> prompt, type:

```
CALC> 5 factorial
120
```

to calculate the factorial number for 5.

## CALC FILES

These files should be in the current directory when running CALC:

- CALC.EXE
- CALC.APP
- HELP.APP

CALC.EXE is the executable file. CALC.APP is a source code file that is automatically loaded by CALC at startup. It typically contains macro definitions. The *hello* macro is defined and executed in the default CALC.APP file. HELP.APP contains text for the help command. It is an ASCII text file that can be edited with any text editor for customization. Both CALC.APP and HELP.APP are optional.

## UNDER THE HOOD

By this time, you are probably more than a little curious about how CALC works. There are two primary portions to the CALC application language, the interpreter and the compiler. Optionally, two additional sections may be present. These are high level CALC macros, such as *factorial*, and user written primitive C/C++ functions you may add to the language. This section addresses only the primary portions.

**THE INTERPRETER** The interpreter used by CALC is named Until, for UNconventional Threaded Interpretive Language. Threaded Interpretive Languages (TIL) are unique in that they are both interpretive and compiled at the same time. The compiler resolves all references and builds a list or "thread" of functions to be executed. The thread is usually a list of low level function addresses. The compiled addresses are placed in a data structure called a dictionary.

There are usually two interpreters in a TIL. The Inner Interpreter that actually executes the functions in the thread by jumping to the addresses compiled in the dictionary. Until uses C's built-in call/return mechanism as the Inner Interpreter.

The Outer Interpreter interacts with the user. The CALC> prompt means the Outer Interpreter is waiting for keyboard input. When a macro name is entered, the Outer Interpreter looks it up in the dictionary, then starts the execution thread. When the thread finishes execution, the Outer Interpreter returns to the prompt and waits for additional input.



There are many ways to implement a TIL. Until takes an unconventional approach in several respects: it is written in C++, uses a noncontiguous dictionary, and has no separate Inner Interpreter.

**THE COMPILER** The compiler compiles references to other macros into the dictionary. Each macro entry in the dictionary has a specific data structure. Information in the dictionary includes the macro name, the primitive macro type, and the list of other macros to execute.

The compiler really determines what the user sees in terms of language syntax. The Callable Application Language Library (CALL) compiler is RPN-based for simplicity. This dictates parameters before the macro/operator name and some other characteristics. There is no reason that any conventional programming language cannot be implemented as a TIL. Microsoft reportedly uses this technology in its line of Quick languages.

The RPN-type compiler is simple compared to traditional compilers such as C or Pascal. There are several reasons for this.

- Simple parsing
- No internal symbol tables
- Doesn't have to generate machine code directly
- There is no difference between interpretive and compiled code in a TIL.

An RPN-based compiler like CALL is simple enough for the average C/C++ programmer to understand, implement, and take advantage of. The CALL compiler is discussed in detail in later chapters.

## OTHER USES

Once you have a simple compiler that you understand and can modify, a whole range of possibilities opens up for new applications. The obvious use is an application language as described in this book. An application language, like CALC, written in C or C++ can easily be incorporated into any application you write. It is likewise easy to add new primitives to tailor the application language to a specific application. One of my favorite ideas for a TIL-based application language is an interactive C++ debugger. Your imagination is the limit of what can be done with an application language once you grasp how the whole concept works.

The rest of this book examines CALC as an application language and discusses both the Until interpreter and CALL compiler in detail. Even if you don't understand everything that goes into the interpreter and compiler, CALC is a ready-made application language that you can add your own specific C++ functions to and use in any C++ program you write.



# Chapter 3

# CALC DESIGN AND IMPLEMENTATION

## INTRODUCTION

This chapter covers the design and implementation of CALC. CALC is an application language that implements a simple calculator language that includes the capability of defining macros with conditionals and loops.

CALC source code is contained in several files. These are:

- UNTIL.H
- CALC.CPP
- UNTIL.CPP
- CALL.CPP
- MATH.CPP
- USER.CPP
- IO.CPP
- PRIM.CPP

## DESIGN

The design of CALC is strongly influenced by CALL. The CALL design implications for CALC include:

- Reverse Polish Notation
- Stack oriented parameter passing
- Integer only

- Space delimited tokens
- Macros must be defined before being referenced.

These constraints are related to the current version of Until. They are not inherent in threaded interpreters. Many are a result of choosing the simplest approach to implementation whenever possible for this book.

Reverse Polish Notation or RPN is a mathematical notation where the parameters occur before the operator. This is how HP calculators work. RPN makes parsing operators almost trivial because nesting is unnecessary. Nested statements require complicated look-ahead parsing. RPN and a stack for parameters go hand in hand. Consider the RPN method for adding two numbers:

```
1 2 +       -->    3
```

Parsing this string is a matter of searching for each space. The "1" is picked up as the first token, converted to a number, and pushed onto the stack. Then the "2" is parsed and pushed on the stack. Finally, the "+" token is extracted and executed. Two numbers are popped off the stack, added together, and the result, three in this case, is pushed back onto the stack. No additional parsing is required. The same is true for compiling.

Now compare the RPN example with the algebraic equivalent:

```
1 + 2       =      3
```

Assuming space delimiters, the first token parsed is "1." It must be stored in a temporary variable. Next, the plus is parsed into a token. Execution of plus looks ahead for the next token, "2" in this case, then adds the two numbers together and places the result in a temporary variable. Parsing complexity is higher because parsing must be done in two stages, as part of each operator in addition to the basic parsing of the line. This is not the only way to parse algebraic expressions.

Take the space delimiter away and algebraic parsing gets even more complicated. Rather than simply using a blank as the delimiter, each character must be examined and a determination made if the character is part of the previous token, or an operator, or part of the next token.

CALL uses stacks and RPN to compile new macros. The compiler is simple enough that the average programmer will have little trouble understanding how it works. Simplicity, in this case, does two things: it adds reliability to the system and influences the operation of CALC. Until is simple enough that you can replace CALL and have a system with a completely different feel.



## CALC OPERATORS

The relatively small number of operators in CALC is done on purpose to keep the system understandable. CALC commands fall into several categories:

- Math operators
- Memory operators
- Stack operators
- Logical operators
- Looping operators
- Miscellaneous operators

The set of operators is relatively small but complete to define a simple programmable language. New operators can be easily added later. The example CALC code snippets in this chapter are contained in EXAMPLE.APP on the source disk.

A common documentation approach for TILs is including a comment about the stack effects of a particular operation. Look at "+" in the list of Math Operators. The stack comment is:

```
{ n1 n2 --- n1+n2 }
```

This describes the operation of adding two numbers together. The Data Stack has two numbers, n1 and n2. The operation is represented by "---" and n1+n2 is the result of the operation. n1+n2 is the new top of the Data Stack. Note that n2 is the top entry on the stack before the operation. Text enclosed in { ... } is considered a comment. Note that the leading blank is required.

**MATH OPERATORS** Since CALC is a calculator applications language, the first step is to define the set of math operations supported. These are:

```
+       { n1 n2 --- n1+n2 }
-       { n1 n2 --- n1-n2 }
*       { n1 n2 --- n1*n2 }
/       { n1 n2 --- n1/n2 }
rem     { n1 n2 --- remainder(n1/n2) }
```

"+" adds the top two numbers on the Data Stack and replaces them with the result. "-" subtracts the top number on the Data Stack from the second number (n1 – n2) and replaces them with the result. "*" multiplies the top

two numbers on the Data Stack and replaces them with the result. "/" divides the top two numbers on the Data Stack and replaces them with the result. That is n1 divided by n2. "rem" returns the remainder of n1 divided by n2 as the top of the stack.

**STACK OPERATORS** A minimal set of stack manipulation macros are part of CALC:

| | |
|---|---|
| dup | { n1 --- n1 n1 } |
| swap | { n1 n2 --- n2 n1 } |
| drop | { n1 --- } |
| rot | { n1 n2 n3 --- n2 n3 n1 } |

"dup" duplicates the top entry of the Data Stack. "swap" exchanges the top two entries. "drop" removes the top Data Stack entry. "rot" rotates the third entry on the stack to the top of the Data Stack. The following examples show the stack before and after each operator:

```
99    |   dup   |   99
      |         |   99

10    |   swap  |   11
11    |         |   10

69    |   drop  |

77    |   rot   |55
66    |         |77
55    |         |66
```

Stack operators can be used either interactively or as part of a macro definition.

**MEMORY OPERATORS** An application language is severely hampered without the ability to define and access variables. Rather than predefine a fixed number of variables, CALC allows the definition of both constants and variables and provides several macros to manipulate them. The CALC memory operators are:

| | |
|---|---|
| var | { --- addr.of.var } |
| const | { --- constant.value } |
| ? | { address --- } |
| <- | { address --- value } |
| -> | { value address --- } |

var and const have both compile time and run time actions. The compile time usage is:



```
var xxxx
1066 const year
```

The first line defines a variable, xxxx, whose address will be placed on the top of the Data Stack when it is referenced. The second line defines the constant year with a value of 1066. When year is referenced, its value, 1066, is left on the stack.

The remaining three operators manipulate variables and constants. "?" types the contents of the address on the top of the Data Stack to the screen. "<-" fetches the value of the address at the top of the stack. "->" stores the value at the address.

Here are some examples of the memory operators:

```
var Battle_of_Hastings       { Define a variable          }
1066 const year              { Define a constant          }
year Battle_of_Hastings ->   { Store 1066 in the variable}
Battle_of_Hastings ?         { Type contents of variable }
Battle_of_Hastings <- 1 -    { Subtract 1 from variable  }
Battle_of_Hastings ->        { Store the new value       }
```

Defining and manipulating variables and constants is relatively simple.

**LOGICAL OPERATORS** Part of programming is logical operations on data. CALC provides the following logical operations:

| | |
|---|---|
| < | { n1 n2 --- truth } |
| == | { n1 n2 --- truth } |
| > | { n1 n2 --- truth } |
| if else endif | { truth --- } |

"<", "==", ">" all compare n1 and n2. TRUE is left on the Data Stack when the comparison is true, or FALSE is left when the comparison is false. TRUE is 1 and FALSE is 0.

A comparison is done first; then the *if else endif* construct uses the truth value on the stack and operates accordingly. The code between *if* and *else* is executed when the stack value is TRUE. Code after the else executes when the stack value is FALSE. The following code examples illustrate how the comparison operators are used:

```
m{ if_example                { --- }
  Battle_of_Hastings <- 1066 ==  { Test var equals 1066}
  if                             { Test condition      }
    the_people_won               { Macro to execute    }
  else                           { FALSE executes here }
```

```
    the_king_won                    { Macro to executo    }
  endif                             { Finish the if       }
}m
if_example                          { Execute the macro   }
```

One final note: comparison operations (<, ==, and >) can be executed at any time. The if ... else ... endif construct is valid only inside a macro definition. m{ and }m define a new macro and are explained below. the_people_won and the_king_won must be defined as macros before being referenced. The code on the disk contains their definitions.

**LOOPING OPERATORS** Another feature of CALC is the ability to loop through a section of code.

| | |
|---|---|
| ( | { n --- } |
| ) | { --- } |

The "(" starts a loop and ")" ends it. The number of times to execute the loop is picked up from the top of the Data Stack. For example:

```
m{ loop_example          { n1 --- }
  10 ( dup                          { Save n1             }
       = cr                         { Type value          }
       1 +                          { Increment counter   }
     )                              { End of loop         }
  drop                              { Clean up stack      }
}m                                  { End macro def       }
1 loop_example                      { Count from 1 to 10  }
```

Other structured looping constructs are possible but are not defined as part of CALC to keep it simple. Loop operators can only be used as part of a macro example.

**MISCELLANEOUS OPERATORS** The remainder of CALC's operators can be categorized as miscellaneous. They are:

| | |
|---|---|
| = | { n --- } |
| macros | { --- } |
| s? | { --- } |
| exit | { --- } |
| { } | { --- } |
| m{ | { --- } |
| }m | { --- } |
| " | { --- } |
| cr | { --- } |
| help | { --- } |



"=" types the number at the top of the Data Stack to the screen. "macros" lists the operators presently defined in CALC. User defined macros are also included in this list. "s?" means stack print. It is a nondestructive type of the contents of the Data Stack. "exit" returns control to the application program. "{" and "}" delimit comments. "m{" begins a macro definition and "}m" ends a macro definition. Finally, " ... " types a string constant to the screen. "cr" types a carriage return to the screen. The following code illustrates each of these operators:

```
m{ a_macro                  { --- }
  " Type a list of Macros" cr       { Type a message      }
  macros cr                         { List all macros     }
  " The year is: " year = cr        { Type value in year  }
  " The stack: " cr s?              { Show the Par. Stack }
  }m                                { End Macro def       }
a_macro                             { Execute macro       }
exit                                { Return to DOS       }
```

Type the following macro definitions before compiling the macro if_example:

```
m{ the_people_won           { --- }
  " The people won!!" cr
  }m
m{ the_king_won             { --- }
  " The king won..." cr
  }m
```

These operators are sufficient to illustrate that the CALC is a full applications language. You may not want to include all of the commands, or you can extend the language by predefining other macros for the user, or you may want to add more specific primitive functions. You have the power to add or subtract features at will.

Primitive macros are also easy to add. A primitive macro is a C++ function that can be called directly from CALC. Most of the operators covered in this chapter are primitive macros. For example, executing the '=' operator calls dot() in PRIMITIVES.CPP. Adding primitive macros is covered in Chapter 10.

## SUMMARY

This chapter covered the CALC application language operators. CALC has full programming capabilities including variables, conditionals, and looping. Additional primitive operators are easy to add to CALC.

Enhancements to CALC are possible. Additional comparison and stack manipulation operators are very easy to add. BCD and floating point math may be added. Infix notation rather than Reverse Polish Notation is also quite doable once CALL is mastered.

All of the pieces are now in place to add our applications language, CALC, to an application program, MORE. The next chapter combines the two.



# Chapter 4

# SAMPLE PROGRAM

## INTRODUCTION

Before getting to the design and implementation of CALC, here is a sample program that will be combined with CALC in a later chapter. MORE.CPP is a simplified version of the MORE program available on many MS-DOS public domain utility collection disks. It is also standard on most Unix systems.

This chapter describes MORE as a typical candidate application program to add an application language to. It is included here for that reason.

## THE MORE DESIGN

MORE accepts a filename from the command line, opens the file, then displays the contents of the file one screen at a time, and closes the file when the end of file is reached.

The first portion of the program includes the appropriate header files. The comments show the functions referenced in each header file.

```
#include <iostream.h>          // cout
#include <fstream.h>           // ifstream
#include <process.h>           // exit
#include <string.h>            // strcpy
#include <conio.h>             // textmode, clrscr
```

The function prototypes are next:

```
void no_more();
void display(char*);
void do_page_break();
int  get_line(char*,int);
```

The input file is declared an input file stream object. input_file is global because it is referenced by more than one function. The filename array is initialized and the size of the input buffer set.

```
ifstream input_file;
char file_name[40];
const BUF_LEN=256;
```

The main function can be divided into several code sections. The first section sets up the file and screen mode. Each line of the input file will be read into "buffer." The maximum line size is 255 characters.

```
main(int argc, char* argv[])
{
     char buffer[BUF_LEN];
     int  status;
```

argc is the command line argument count plus one for the filename. Thus argc will be two when a filename is included. An informational message is printed and the program exits when the number of arguments is incorrect.

```
if(argc != 2){
     cerr << "USAGE: MORE filename \n";
     exit(-1);
}
```

The next lines open the input file and associate the file object, input_file, with it. argv[1] contains a character pointer to the first argument string.

```
strcpy(file_name,argv[1]);
input_file.open(file_name,ios::nocreate);
if (!input_file){
     cerr << "Cannot open input file: " << argv[1]
<< "\n";
     exit(-1);
}
```

The final two lines setting up the program initialize the PC screen for output. The textmode is set and the screen cleared. Change the textmode appropriately if the PC has a different type of monitor attached. These two function calls are Borland/Turbo C++ specific.

```
textmode(LASTMODE);
clrscr();
```

The program is now ready to begin the primary loop. The for(;;) loops forever. The loop will not exit unless a goto or call to exit( ) is executed. The input file is read via get_line( ) and each line output to the screen via display( ) until an end of file condition is encountered. The function



no_more( ) closes the file and calls exit( ), effectively jumping out of the loop.

```
for(;;){
     status = get_line(buffer,BUF_LEN);
     if(status){                    // 0 means eof
          display_line(buffer);
     }else{
          all_done();               // done, so quit
          }
     }
}
```

MORE.CPP is divided into multiple functions following good programming practice.

```
void all_done()
{
     input_file.close();
     exit(0);
}
```

The next function in MORE.CPP is get_line( ). It reads a line from the input file into the character array pointed to by *line*. Both filling the buffer and end of file are tested for. The test for a full buffer is implicit in the for loop because it stops when the maximum buffer length is reached. End of file is tested by placing the read in an if statement and changing the status when end of file occurs.

Control flow jumps out of the for loop when a newline character, '\n', is detected. A null, '\0', is appended to the end of the buffer to mark the end of the line. The value returned in status indicated whether end of file was reached on this read or not.

Reading a character from the input file is performed by calling the method get( ) for the input_file object. Remember that input_file is a member of class ifstream. Thus, you expect to find a function named ifstream::get( ) in the public portion of the class statement for ifstream.

```
int get_line(char* line,int max_len)
{
     int  status=1;
     char ch;

     for(int i=0;i<max_len;i++){
          if(!input_file.get(ch))   status=0;
          *line++ = ch;
          if(ch == '\n'){
               break;
          }
```

```
        }
        *line++ = '\0';
        return(status);
}
```

The next function, display_line( ), outputs a line to the screen. display_line( ) counts the number of lines written to cout. When a page break is detected, the function do_page_break( ) is called to handle the details.

```
void display_line(char* buffer)
{
        static int line_ct = 0;
        const LINES_PER_PAGE = 20;

        cout << buffer;
        line_ct++;
        if(line_ct > LINES_PER_PAGE){
                do_page_break();
                line_ct = 0;
        }
}
```

The final function in this program is do_page_break( ). Both cout and cin are used here. Cout queries the user whether or not to continue. cin reads the response from the user. 'n' is the only value tested for. Any other key is treated as a 'y.' When you run MORE.CPP, notice that Enter must also be pressed before the character is accepted.

The screen is cleared and a heading of the filename is typed to prepare the PC screen for the next page of output.

```
void do_page_break()
{
        char answer;

        cout << "\n" << "Continue (Y or N)? ";
        cin >> answer;
        if(answer == 'n' || answer == 'N'){
                exit(0);
        }
        clrscr();
        cout << "FILE: " << file_name << "\n\n";
}
```

## SUMMARY

MORE.CPP is a simple C++ application program. It is used in the next chapter to add the CALC applications language.



# Chapter 5

# CALC AND MORE

## INTRODUCTION

This chapter ties all of the pieces of the book together from the standpoint of adding an application language to a program. It illustrates how to combine an application language (CALC) created by Until and a stand-alone program (MORE). The necessary steps for building the combined program are discussed. The new program is MORECALC.CPP.

## THEORY OF OPERATION

An application program should be interactive to take best advantage of CALC. A pause when the program is soliciting user input is the logical place for the user to invoke the application language. MORE.CPP waits for user input at the end of each screen. The original code read a character from the keyboard and tested it for 'Y' or 'N' to indicate whether to continue or exit. This is the spot to insert the call to CALC.

There are only two steps to set CALC up and start an interactive session.

- Call outer to start CALC
- Compile CALC and link with the application program

Type **exit** to leave CALC and return to the application program. CALC can be invoked any number of times during a run. The steps listed to call CALC apply to any application language developed using Until.

## CHANGES TO MORE

This section identifies the changes to combine CALC with MORE. Only the changes will be discussed here. The full program was discussed in Chapter 4.

A new function prototype must be added to the existing function prototypes at the beginning of the program.

```
void outer();
```

All of the code changes to MORE are isolated to a single function, do_page_break( ). The following code is the new version:

```
void do_page_break()
{
     char answer;

     cout << "\n" << "Continue (Y or N or C)? ";
     cin >> answer;
     switch(answer){
     case 'Y':
     case 'y':
               break;
     case 'N':
     case 'n':
               exit(0);
               break;
     case 'C':
     case 'c':
               outer();
               break;
     }
     clrscr();
     cout << "FILE: " << file_name << "\n\n";
}
```

The changes are a new switch statement replacing the if test to exit in the original version. Type 'C' at the continue prompt to invoke CALC when running MORECALC.

The modified file is MORECALC.CPP. Compiling it is accomplished using the project file MORECALC.PRJ which is accessed via the Project pull-down menu in Borland C++. The result is MORECALC.EXE. Be sure that the compiler is set to generate a compact or larger model program.



## SUMMARY

This chapter showed how to combine an Until application language with a stand-alone program. CALC macros can be written and loaded by placing the commands in CALC.APP.

The goal of easily adding an applications language to a stand-alone program was demonstrated. The interactive execution of Until can be utilized to tie almost any set of functions into another program. The next chapter begins delving into the mysteries of Until.

# Chapter 6

# UNCONVENTIONAL THREADED INTERPRETIVE LANGUAGE

## INTRODUCTION

The first chapters introduced the concept of an application language and a simple implementation in the form of CALC. CALC was combined with a stand-alone program, MORE, to illustrate the ease with which this can be done. The remaining portion of this book delves into the underlying software technology, Until, that makes creating an application language such an easy task.

This chapter introduces details of the *UN*conventional *T*hreaded *I*nterpretive *L*anguage (Until). Until is the threaded interpreter used in CALC. This chapter covers the overall design considerations for Until, an overview of threaded interpretive languages, and some of the things that make Until unique.

The first thing to point out is that you do not have to totally understand how Until works to use it. On the other hand, the more you understand about how it works, the more customization you can do to mold the resulting application language to your own liking.

Until macros are logically equivalent to functions in C++. Macros are stand-alone units of code that can be individually referenced. Some TILs use the term "word" in the same context that this book uses macros.

## DESIGN CONSIDERATIONS

The design considerations and tradeoffs for Until were carefully considered. The primary design goals for Until were:

- A portable TIL
- Small "macro" language that is callable by other C/C++ programs
- Easy-to-add C++ primitives
- Easy-to-implement application languages

These design considerations translate into goals. The ease constructing CALC shows that Until has met its goals. Each of these items is discussed in detail next.

**WRITE A TIL IN C/C++** There are many reasons for writing Until in C/C++. Some of my reasons are:

- Portability
- TILs are relatively simple
- TILs are neat
- Experience with other TILs

Portability is often important. A TIL written in portable language, such as C or C++, is very portable among different computer systems. TILs are typically written in assembly language. This gives excellent speed but lacks portability. A TIL written in C or C++ does not have the speed potential of one written in assembly language but it is very portable.

TILs are relatively simple when compared with conventional language compilers. Conventional compilers, such as C++, are large programs that parse and translate a C++ program from source code form into the native assembly language of the computer being used. The Turbo C++ .EXE file is over 800k plus overlays. Each line of source is parsed into individual tokens. This is complicated by operator nesting. The compiler keeps symbol tables of variables and function names. Code generation and optimization are also very complex.

A TIL uses very simple parsing and requires no symbol tables. It can be written without the need for nesting by using postfix notation rather than conventional infix notation. It may or may not generate code directly. Until generates addresses of C++ functions to execute. (Until has a separate compiler module named CALL that is described in Chapter 9.) Some commercial languages that are TILs do generate machine code.



TILs are an interesting cross between a compiler and an interpreter. I first discovered TILs in the form of Forth in the December 1980 issue of *Byte Magazine*. They have fascinated me ever since. I have used both Forth and another TIL named STOIC on several projects over the years since.

I see the value in keeping a TIL among my software development tools. I have been using Forth for the past few years at work. As the project moves toward C and C++, I want to keep the interactive nature of a TIL in new programs, even though C++ is not an interactive language. This is a primary reason for writing Until.

**CALLABLE MACRO LANGUAGE** One of the primary goals for Until is a TIL that is callable by any C++ program. This capability puts the ability to create and use an application language in the hands of all C++ programmers. This opens up a great many possibilities not normally available in C or C++. If your C++ does not have a source code debugger, simply put calls to Until into a program instead of time honored printf statements for debugging. You end up with an interactive debugging environment that is quite productive during the debug phase of system development.

**EASY TO ADD PRIMITIVES** A number of Forth implementations in C exist. Some of them are quite good. They all suffer from a common problem: they are difficult to add new primitive C functions to. Likewise, all were intended to be stand-alone programs rather than being called from other programs. Until takes exactly the opposite approach. Adding new primitives is simply a matter of writing a new C++ function and adding an entry to a table. The C++ function must follow a specific set of interfacing rules. The interfacing rules are covered in a later chapter.

**IMPLEMENT AN APPLICATION LANGUAGE** The final design goal for Until is providing an easy way to add an application language to any C++ program. Application languages add a professional air to any program. There is great potential for their use in many programs.

## BACKGROUND

This section provides some background on threaded interpreters. The premier TIL is Forth. Until is based largely on concepts found in Forth, with a few twists.

**FORTH** The Forth language was invented in the early 1970s by Charles Moore, while at the National Radio Astronomy Observatory in Kitt Peak, Arizona. The very first Forth implementation was on an IBM 1130 computer. It only allowed five character names. So when Moore developed what he considered the first fourth generation language, "fourth" was shortened to "forth" because of the computer it ran on. Moore and Elizabeth Rather went on to form Forth, Inc. The company is still in business as one of the premier Forth vendors.

Forth's strength remains in real-time control and embedded systems to this day. Software for several Space Shuttle experiments are written in Forth. A number of successful commercial products have been written in Forth over the years:

- EasyWriter I — The first word processor for the IBM PC
- Rapid File — Flat file database system from Ashton-Tate
- VP-Planner — Spreadsheet, so good that Lotus sued them out of business
- Zoomracks — Shareware free format text database system
- SAVVY — Early CP/M database system
- Some early Spinaker games for Apple II
- GE Locomotive Test Stand — Expert Systems
- Canon CAT — Dedicated word processor
- Early Peachtree accounting packages
- Many Atari Games

In 1973 the Forth Interest Group (FIG) was formed to help spread the language on small microcomputers. A group of FIG members implemented what became known as FIG Forth on over a dozen computer systems and distributed program listings at a nominal cost. Several current Forth vendors got their start as a direct result of FIG's efforts. FIG Forth became the defacto Forth standard for several years. The Forth community has created two generally accepted standards since that time, Forth-79 and Forth-83.

As this is written, Forth is currently going through the process of becoming an ANSI standard language. The ANS ASC X3/X3J14 Technical Committee is responsible for this effort.

**SOMETHING BORROWED; SOMETHING NEW** Until borrows significant ideas from Forth. There are also many departures from traditional Forth implementations. Some of the ideas in common with Forth include:



- Dictionary data structures
- Interpreters
- Both primitive and high level words/macros
- Data Stack to pass parameters
- Reverse Polish Notation (RPN) syntax
- Separate RPN-type compiler

Many Until variables and function names are taken directly from Forth. The dictionary is a data structure designed to support the threading action of the TIL. The dictionary is discussed in detail in Chapter 7. Until is different in that Forth generally is implemented using a contiguous dictionary. Until allocates space for each new macro as it is defined. This prevents running into the 64k array limit in some C/C++ compilers. Forths written in C typically allocate a single large array for the dictionary.

Until uses C's built-in function call/return mechanism for threading from macro to macro. Forth has a separate Inner Interpreter for threading. Until and Forth both have an Outer Interpreter to handle user interaction.

The Until compiler is separate from the rest of the system. The RPN-type compiler in Until resembles a Forth compiler. Until is designed so the RPN-type compiler can be removed and any compiler you may want to write can be substituted.

The final departure from traditional Forth is the choice of implementation language. Forth has been implemented in C several times over the years. Each is an almost exact implementation of the Forth model in C. The premier version is CForth by Bradley Forthware. It is usable and runs well on several different computers including PCs, Unix systems, and workstations. Other notable implementations include TILE and a FIG Forth in C; both are public domain or shareware and specific to Unix. Commercial Forth implementations are generally written in the native assembly language of the target system.

## THREADING TYPES

Threaded interpreters can utilize several different types of threading. These include:

- Indirect Threaded Code
- Direct Threaded Code
- Token Threaded Code

- Subroutine Threaded Code
- Segment Threaded Code
- Switch Threaded Code

A TIL can be implemented using any of these threading techniques. Forth traditionally uses indirect threading. The advantages to each type of threading is discussed in the next several paragraphs. Program threads are traversed by the interpreter. (The Until interpreter is discussed in detail in Chapter 8.)

*Indirect threaded code* is the traditional method of threading used in Forth implementations. Words are comprised of a header and a list of word Code Field Addresses (CFA). CFAs point to primitives to process the type of word. Indirect is the slowest of the four threading types but generally produces the most compact code in terms of memory.

*Direct threaded code* is a variation of indirect threaded code. Words are still a list of CFAs. The difference is the CFA contains either inline code or a call to a primitive subroutine. Direct threaded code is faster than indirect threaded because one level of code indirection is removed.

*Token threaded code* has the primary advantage of being easily relocatable. Words consist of a header and list of token pointers. A table of tokens and code addresses is maintained. If 8-bit tokens are used, token threading produces the smallest code. It is generally slow because of the extra level of indirection for accessing the token table to get the address to execute.

*Subroutine threaded code* is the fastest of the threading techniques. A word is a list of subroutine CALLS to lower level words. This approach takes more memory than other threading methods and generally needs two hardware stacks to implement.

*Segment threaded code* is a method of threading devised to cope with limitations imposed by the INTEL 80X86 segmented architecture. Each new macro is started on a new segment boundary. This wastes some space (segments are 16 bytes) but allows the dictionary to be as large as memory rather than restricted to 64k.

*Switch threaded code* is used by several versions of Forth written in C. The inner interpreter is a giant switch statement with all primitives being cases. This is very efficient in C and general opinion is that switch threaded code results in a fast interpreter.



## UNTIL DESIGN

Until uses a modular design. This allows substituting modules as necessary for the application. The Until modules are:

- Outer Interpreter
- Compiler
- Core primitives

Each module is covered in detail in its own chapter.

# Chapter 7

# UNTIL DATA STRUCTURES

## INTRODUCTION

Data structures are the heart of every system; Until is no exception. This chapter identifies and discusses the primary data structures used internally by Until. The Until data structures are defined in the file UNTIL.H.

Until's data structures may be divided into three groups:

- Stacks
- Dictionary
- Other

Each is addressed in a separate section.

## STACKS

A stack is a last-in first-out (LIFO) memory array. Stacks can be implemented either in hardware or in software. Some processors have direct support for one or more stacks.

Virtually every programming language uses stacks for passing arguments between functions or subroutines. The difference is that the programmer rarely has access to the stacks. The programmer not only has access to the stacks in Until; a stack is dedicated to passing arguments between macros.

Many TILs are implemented with two stacks, the Parameter or Data Stack and the Return Stack. The Data Stack is equivalent to the internal stack described above for passing arguments between functions. There is a full set of stack manipulation macros associated with the Data Stack.

A stack cell is manipulated by primitive Until macros. Until treats a stack cell as a long integer number by default. The width is 32 bits.

The Return Stack in a typical TIL is used for two purposes. It holds addresses of the next macro for the Inner Interpreter to execute, so it must be wide enough to hold an address. The second purpose is for holding loop indices. Until uses the Return Stack only for loop indices.

**THE DATA STACK** The Data Stack is an array of 32-bit integers used to pass parameters between macros. Numbers typed at the keyboard are automatically pushed onto the Data Stack.

An RPN-based stack oriented language like Until needs a set of stack manipulation primitive functions. Most, including HP calculators, include some of all of the following capabilities:

| | |
|---|---|
| dup | Duplicate the top number on the stack |
| drop | Remove the top number from the stack |
| swap | Exchange the top two numbers on the stack |
| rot | Move the third number to the top of the stack |
| pushsp | Push a number onto the top of the stack |
| popsp | Move the top number of the stack to a specified variable |

Until stores all data types on the Data Stack by casting the value to long integer before pushing it onto the stack. When numbers are popped off the Data Stack, the calling function is responsible for casting from long integer into the proper type.

*pushsp* and *popsp* are internal low level functions that are automatically called when necessary. The user never calls either primitive directly.

The other part of implementing a stack data structure is a stack pointer. A global variable, SP, is Until's stack pointer. It is incremented when a number is pushed onto the stack and decremented when a number is popped off. Until tests for both stack over and underflow.

The following code defines the Data Stack:

```
constant int PSTACKSIZE = 64;
long pstack[PSTACKSIZE];
long SP;                          // stack pointer
```

The stack depth can be increased by changing PSTACKSIZE. Tests for stack overflow references PSTACKSIZE.



**THE RETURN STACK** The Return Stack is a special purpose stack. Until uses it for storing loop indices. The maximum number of times a loop is to be executed and the current loop index are pushed onto the Return Stack at the beginning of a loop.

The loop index is incremented and tested for greater than the maximum loop count at the end of each loop execution. If the exit condition is not met, the loop continues. Otherwise, it exits.

The primitive Return Stack manipulation macros are:

| | |
|---|---|
| rpush | Push a number on the stack |
| rpop | Pop a number from the stack |
| rfetch | Get the top number from the stack without changing the top value |

A simple extension to add to Until is allowing the user to push temporary values onto the Return Stack. This has far ranging implications that should be understood before providing user access to the Return Stack.

There is a stack pointer dedicated to the Return Stack. It is a global variable named RP. The following code defines the Return Stack:

```
const long RSTACKSIZE = 32;
long rstack[RSTACKSIZE];
long RS;                          // Return Stack
pointer
```

## DICTIONARY

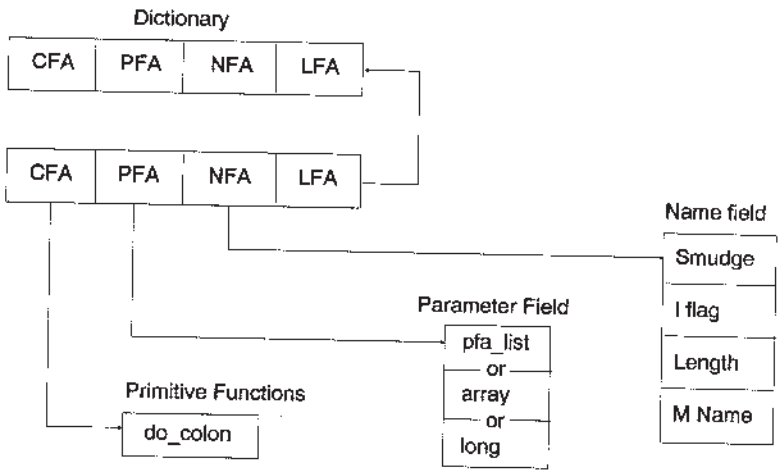
The second major data structure in Until is the dictionary. All Until macros are compiled into the dictionary. The dictionary is a linked list structure. The entry for each macro is a DictHeader.

The typical TIL uses a linear dictionary; Until does not. Until's dictionary is a linked list of pointers. Space for each new macro is allocated dynamically as it is compiled.

The fields in each dictionary headers are:

- Name Field Address (NFA)
- Link Field Address (LFA)
- Code Field Address (CFA)
- Parameter Field Address (PFA)

All are used during compilation. The following illustration depicts a dictionary entry.




The PFA can be one of several data types. A C++ union is used to define the PFA. The following code defines a dictionary entry:

```
struct DictHeader {
     void (*CFA)();      // ptr to primitive function
     union pfa_type{
          DictHeader **Waddr;
          long       lvalue;
          char       cvalue;
          long       *l_ptr;
          char       *c_ptr;
     } PFA;        // ptr to list of words to execute
     NameField  *NFA;  // ptr to name string
     DictHeader *LFA;  // points to next word in
                          dict
};
```

A new dictionary entry is allocated each time a constant, variable, or macro is defined. The minimum size allocated for a dictionary entry is:

```
Dictionary Header        32 bytes
Name field               32 bytes
                         --
                         64 bytes plus length of the
                            name
```

Assume the average name is eight characters long. So, each macro definition takes an average of 72 bytes. This sounds like a lot. The other



side of the coin is that a reference to a macro only takes four bytes. Calls in C or C++ take a minimum of six bytes each.

**NAME FIELD ADDRESS** The NFA is a pointer to a name field structure. The structure contains four fields:

- Length of the name text
- I or Immediate Flag
- Smudge Flag
- The macro name text

Both the interpreter and compiler look up parsed macro names in the dictionary. The name field provides the name information that is used in the dictionary search. Here is the structure that defines the name field:

```
struct NameField {
     int  len;
     int  immediate;
     int  smudge;
     char *name;
};
```

**Length of the Name Text** The number of characters in the macro name is stored in the name field. Forth strings are counted rather than null terminated. Until uses counted strings internally, hence the length field. Counted strings speed up searching the dictionary because the full string compare on name is only required if the lengths match first.

**Immediate Flag** The Immediate Flag causes a macro to be treated as type immediate. Immediate macros execute during the compilation process rather than being compiled into the new definition. CALL only allows primitive functions to be declared immediate. The Immediate Flag has no effect during interactive execution.

The comment operator, '{', is immediate. When a '{' macro name is parsed and a match found in the dictionary, the compiler tests the Immediate Flag before compiling the macro address into the dictionary. The macro is executed if the flag is set. That is how the comment operator works inside a macro definition. All of the compile-only macros, such as *if ... else ... endif* are immediate.

**Smudge Flag** The Smudge Flag provides a way to hide a definition in the dictionary. It is set during the compile process for the macro currently being compiled. The compiler searches the dictionary and flags any duplicate definitions found. The flag is cleared as the last compilation step.

The Smudge Flag effectively prevents accidental recursion. CALL, as defined in this book, does not support recursion. Recursive macros are possible with the addition of the proper primitives.

**Macro Name Text** The actual text of the macro name is stored in this field as a pointer to a string. The name string is allocated as one of the first steps in compiling a macro. The string is also null terminated so it can be manipulated using built-in C string functions.

**LINK FIELD ADDRESS** The LFA points to the previous dictionary header. The LFA of the first word in the dictionary contains an LFA of 0. The LFA is used to step through the dictionary when searching for a macro name. This is the "thread" that ties the macros together in the dictionary.

**CODE FIELD ADDRESS** The CFA points to the primitive C++ function to be called when this macro executes. It is defined as a void pointer. The CFA is the execution "thread." The CFA always points to a primitive C++ function.

**PARAMETER FIELD ADDRESS** The PFA contains the parameters for the macro. The contents are specific to each primitive macro type. The PFA for both variables and constants contains a long integer. It contains a list of macro addresses to execute for a macro definition. Knowledge about the contents of the Parameter Field is the responsibility of the primitive.

## OTHER STRUCTURES

Until uses several other minor data structures. This section identifies them. The data structures discussed here are:

- Terminal Input Buffer (TIB)
- Scratch pad area (PAD)
- Temporary PFA (PFA_LIST)
- Miscellaneous Pointers

Many TILs have some equivalent to these data structures. Each is a pointer or array depending on usage.

**TIB** The Terminal Input Buffer is the character array that user input is read into by the interpreter. It is a maximum of 255 characters in length. The current offset is stored in the global variable *IN*. The user cannot access TIB directly in CALC. The definition for TIB is:



```
const int TIBSIZE = 256;
char  tib[TIBSIZE];
```

**PAD** PAD is a scratch pad area that is used by low level primitive macros. Its primary purpose is for building temporary strings during compilation. The macro '"' uses PAD as a work area too. The user does not have access to PAD in this system. PAD is defined as:

```
const int PADSIZE = 128;
char  pad[PADSIZE];
```

**PFA_LIST** The PFA_LIST field is unique to Until. It is a temporary array of macro pointers sized for 100 entries. That means a macro can reference a maximum of 100 other macros. As a macro is compiled, references are added onto the end of PFA_LIST. Memory for the final compiled list is allocated at the end of compilation for the actual size needed to hold the real PFA entry. This approach makes it relatively easy to use only the needed memory for a definition rather than always allocate a fixed maximum. PFA_LIST can be expanded by increasing the size of the constant PFA_SIZE. The following code segment defines PFA_LIST:

```
const int PFA_SIZE = 100;
DictHeader *pfa_list[PFA_SIZE];
```

**MISCELLANEOUS POINTERS** Both the compiler and interpreter use pointers to keep track of the state of the TIL. The primary global pointers are:

- Dictionary Pointer (DP)
- Instruction Pointer (IP)
- Word Address (WA)

Most TILs have equivalent pointers. Usage of the miscellaneous pointers is discussed again in later chapters as they are referenced.

**Dictionary Pointer** DP is the pointer to the most recently defined macro. It is the starting point for dictionary searches. DP is used both by the compiler and interpreter. CALL modifies the Dictionary Pointer each time a new macro definition is added to the dictionary. The interpreter uses DP as the start of the linked list when looking up a macro name.

**Instruction Pointer** The IP is logically the instruction counter for the interpreter when it executes a high level macro definition. A high level macro is one created using *m{* and *}m*. The IP is also used by the compiler when building the PFA_LIST.

**Word Address** WA is the pointer to the current macro in the dictionary. WA is set when the interpreter finds a match in a dictionary search. The WA is used to get the CFA to execute. The Word Address is used both by the interpreter and the compiler. WA is typed DictHeader*.

## SUMMARY

Until uses a variety of data structures to implement a threaded interpreter. These include stacks, a macro dictionary, and several lesser data structures. The general data structures are borrowed from Forth. Their implementation in Until does not resemble any Forth implementation that I know of. This is especially true of Until's dictionary layout. All of the data structures identified in this chapter are global.

These data areas are usually accessible in other TILs, including Forth. User access to the underlying TIL data structures is restricted because it is not needed for CALC.

The discussions about Until's data structures provide a starting point for the discussion of Until's interpreter in the next chapter.



# Chapter 8

# THE UNTIL INTERPRETER

## INTRODUCTION

Threaded Interpretive Languages traditionally have two interpreters. The first is a low level or Inner Interpreter. The higher level or Outer Interpreter interacts with the user. This chapter discusses Until's Inner and Outer Interpreters.

## INNER INTERPRETER

The Inner Interpreter is the heart of a TIL. While the Outer Interpreter deals with the user, the Inner Interpreter interacts with compiled macro definitions in the dictionary. The Until Inner Interpreter can also be referred to as an address interpreter.

Until does not have a separate Inner Interpreter because it uses subroutine threading that takes advantage of C's built-in function call/return mechanism to thread from macro to macro. Other threading techniques require separate code to implement the Inner Interpreter.

Even though there is no separate inner interpreter in Until, discussions of TIL internals would not be complete without including how a "generic" inner interpreter works. The remainder of this section does just that.

If the inner interpreter is at the heart of a TIL, the macro NEXT is the heart of the inner interpreter. NEXT should always be a machine language primitive. Execution speed is critical in NEXT. An inefficient implementation of NEXT will make a TIL slow.

The generic inner interpreter described here uses indirect threaded code. Macros in indirect threaded code have a header and a list of Code Field

Addresses (CFAs). The indirect threaded code architecture uses four internal registers. These are:

- IP - Interpretive pointer
- WP - Macro pointer
- RP - Return pointer
- SP - Stack pointer

Two of these (the SP and RP) are memory pointers in many implementations, especially on small 8-bit processors because of limited hardware registers. The use for each of the pointers is described below:

**IP** contains the CFA of the next macro to execute.

**WP** contains the CFA of the current macro to execute.

**RP** is the pointer to the Return Stack. It contains the CFA of the macro to return to when the current level of interpretation is completed.

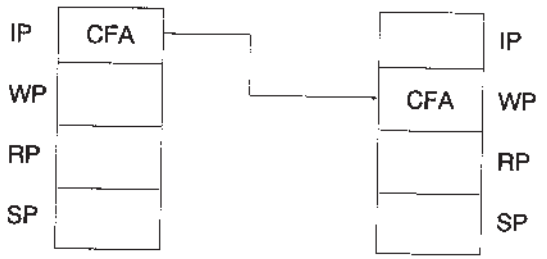
**SP** is the pointer to the current entry in the Data Stack.

The inner interpreter usually consists of four routines:

- NEXT
- EXECUTE
- DOCOLON
- EXIT

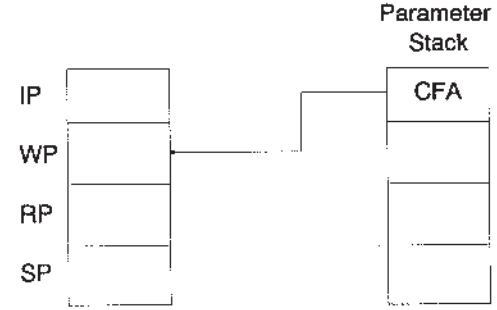
These macros are all machine language primitives that interact with the four TIL registers described to interpret and execute TIL macros.

**NEXT** assumes the CFA of the next macro to execute is in IP. It moves the contents of IP into WP. Some versions of NEXT also automatically increment IP. The macro pointed to by WP is then executed.

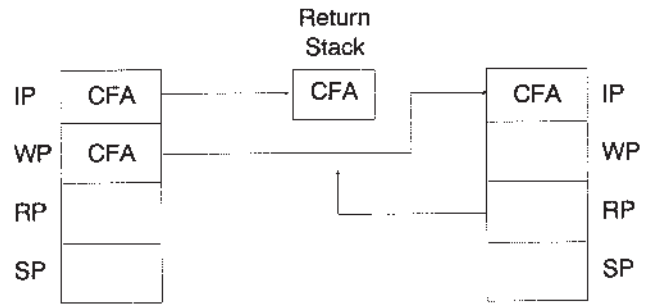




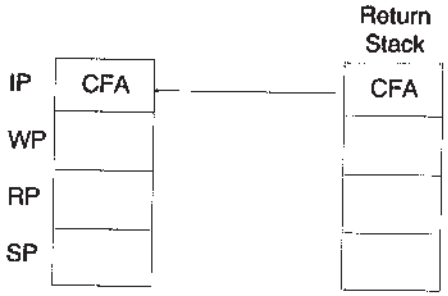

**EXECUTE** expects the CFA on the Parameter Stack. It loads the CFA into WP. The macro is then executed either with a jump (JMP) to WP or a subroutine call (JSR). EXECUTE is called RUN in some early TILs.




**DOCOLON** assumes that the CFA of the macro to return to is in the IP and the CFA of the current macro to execute is in WP. The CFA in WP is converted to a Parameter Field Address (PFA). The contents of the IP is pushed onto the Return Stack, modifying RP. The CFA contained in the PFA is moved into IP, then NEXT is invoked. DOCOLON is the primitive routine whose address is compiled into the CFA of a high level macro. DOCOLON is also named NEST in some older TILs.




**EXIT** is the last macro executed at the end of a high level macro. It pops the CFA off the Return Stack into IP.




**THE INNER INTERPRETER AT WORK** The description of the inner interpreter pieces sounds complicated. Each macro taken individually is relatively simple. The complicating factor is that they are all independent and can be called at any time by each other. So keeping straight who does what to whom can be confusing. The following example of high level and description of the inner interpreter at work should clear up inner interpreter operation.

We will follow the inner interpreter execution of the following macros:

```
m{ HELLO              " Hello " }m
m{ WORLD              " World" }m
m{ HELLO-WORLD        HELLO WORLD }m
```

Now assume that some other TIL macro invokes HELLO-WORLD. Here is how the inner interpreter threads from macro to macro, executing HELLO-WORLD.

```
MACRO                          ACTION
NEXT                    Called to start HELLO-WORLD
     DOCOLON            CFA of HELLO-WORLD is executed
                        (DOCOLON)
     NEXT               Called to execute next macro (HELLO)
          DOCOLON       CFA of HELLO is executed (DOCOLON)
          NEXT          Called to execute next macro (")
               ...      Execute primitive "
          EXIT          Exit macro (HELLO)
     NEXT               Execute next macro (WORLD)
          DOCOLON       CFA of WORLD is executed (DOCOLON)
          NEXT          Called to execute next macro (")
               ...      Execute primitive "
          EXIT          Exit macro (WORLD)
     NEXT               Execute next macro
     EXIT               Exit macro (HELLO-WORLD)
NEXT                    Execute next macro from calling macro
```

The indentions under the heading **MACRO** show the nesting of macro execution. Each indention implies a Return Address being pushed or



popped off the Return Stack. Notice how often NEXT is called, especially when high level macros are involved.

## OUTER INTERPRETER

The Outer Interpreter is Until's interface with the user. Simply stated, the Outer Interpreter waits for the user to type a line at the keyboard. Each word in the input line is parsed, looked up in the dictionary, and executed.

The Outer Interpreter has several functions to perform at the lower level. These include:

- Initialization/startup
- Print the prompt
- Search the dictionary for a macro
- Execute a macro
- Convert digits to a number
- Warm start on an error

Efficiency is not the prime concern in the Outer Interpreter. Most of its processing time is spent waiting for user input. Reliability, on the other hand, is critical.

**INITIALIZATION/STARTUP** The first call to Until initiates several actions:

- Initialize stacks and pointers
- Display the copyright screen
- Open CALC.APP
- Execute the contents of CALC.APP
- Close CALC.APP
- Begin the Outer Interpreter loop

The commands in CALC.APP are executed only the first time Until is called from an application program. Subsequent calls do a warm start only.

**PRINT THE PROMPT** The function prompt( ) simply prints the Until prompt, "CALC>." The prompt string can be changed by changing the value in the global variable PROMPT.

**SEARCH THE DICTIONARY FOR A MACRO** As each word of the input line is parsed, the dictionary is searched for a match with an

existing macro name. The result of the search is flagged for later use by the Outer Interpreter.

**EXECUTE A MACRO** When a dictionary search is successful, the dictionary address of the macro is returned in WA. The address of WA->CFA is executed. This is essentially a call to the Inner Interpreter.

**CONVERT DIGITS TO A NUMBER** If the parsed word from the input stream is not the name of a macro in the dictionary, an attempt is made to convert it to a number and push it onto the stack. Frequently used numbers, such as one, two, and three, are defined as constants to speed up compilation because the definition will be found without searching the entire dictionary, then requiring a numeric conversion. An error is flagged if the conversion fails.

**WARM START ON ERROR** When an error occurs, such as a macro not being found, error processing in the form of a warm start is performed. Warm start processing includes typing a message to the user and resetting the stacks. The Outer Interpreter picks up at the beginning of the main loop.

## OUTER INTERPRETER CODE

The operation of the Outer Interpreter is discussed using two forms, pseudo code and the actual outer( ) function. The following pseudo code logically describes the operation of Until's Outer Interpreter:

```
BEGIN
     INITIALIZE STACKS
     BEGIN
          WAIT FOR TERMINAL INPUT
          BEGIN
               PARSE TOKEN
               SEARCH DICTIONARY
               IF (TOKEN FOUND)
                    EXECUTE MACRO
               ELSE
                    CONVERT TOKEN TO A NUMBER
                    IF (CONVERSION ERROR)
                         ABORT
                    ELSE
                         PUSH NUMBER ON STACK
                    THEN
               THEN
               PRINT PROMPT
          UNTIL (END OF INPUT LINE)
     UNTIL (ERROR)
FOREVER
```



This is a conceptual representation. The Inner Interpreter is invoked indirectly by the EXECUTE MACRO statement.

The Outer Interpreter is function outer( ). It is amazingly simple. The operation of each step identified in the pseudo code is identified and discussed in the following paragraphs.

```
void outer()
{
     static started;
     int    found;

     if(!started){
          started = TRUE;
          startup();
     }
     QUIT = 0;
     for(;;){
          if(QUIT){
               return;              // Initialization
          }else{
               warm();
          }
          do{
               pushsp(BLANK);
               word();      // get next word from
                            // input stream
               minus_find();
               found = popsp();
               if(found){  // found the macro
                    WA = (DictHeader*) popsp();
                    (*WA->CFA)();
               }else{       // Word not found, so
                            // try to turn
                    drop();     //   it into a
                                //   number.
                    pushsp((long)pad);
                    number();  // Convert it to int.
               }
               gdot_s();        // Type stack
          }while(!ABORT);
          ABORT = 0;
     }
}
```

All of the code for the startup/initialization is in the function startup( ). The static variable *started* is tested. It is set and startup( ) called on the initial call to outer( ). The startup( ) is skipped on all other calls.

```
if(!started){
     started = TRUE;
     startup();
}
QUIT = 0;
```

The started flag allows multiple calls to outer( ) from an application program. Macros compiled in the previous invocation remain defined.

A warm start is performed under two conditions. First, when the Outer Interpreter is called after the first time. Second is on an Until error such as a stack underflow.

```
if(QUIT){
     return;                    // Initialization
}else{
     warm();
}
```

Executing the warm( ) function clears the stacks and resets the pointers.

Searching the dictionary is performed by the following code:

```
pushsp(BLANK);
word();
minus_find();
found = popsp();
```

word( ) parses the next word from the input stream. The delimiter character is the top entry of the Data Stack. The call to pushsp(BLANK) pushes a blank onto the top of the stack. The parsed word is returned in PAD. minus_find( ) performs the dictionary search for the parsed word and returns truth on the Data Stack.

The address of the DictHeader is left on the top of the stack by minus_find( ) whenever the dictionary is searched and a match found.

```
WA = (DictHeader*) popsp();
(*WA->CFA)();
```

WA is loaded, then the CFA called to execute the macro.

Converting the word to a number is the final step if the dictionary search fails.

```
pushsp((long)pad);
number();
```

Number calls the C++ library function atol( ) to perform the actual numeric conversion.



## MACRO EXECUTION

It is time to examine what happens when a macro executes. The CFA of every macro points to a primitive C++ function. The valid C++ functions are determined by the compiler. If you force a garbage address into a CFA and then try to execute it, the interpreter merely does what it is told and tries to execute the address. The computer is likely to crash when this happens.

Several classes of macros are defined by the CALL compiler. These are:

- Variable
- Constant
- High level macro
- Primitive macro

The classes are a function of the compiler, not the interpreter.

The macro class is important because the class determines the content of the PFA. This is, in fact, very important as we shall see.

| CLASS | CFA | PFA |
|---|---|---|
| Variable | do_variable( ) | Long Integer |
| Constant | do_constant( ) | Long Integer |
| Primitive Macro | Any function | Depends on the primitive |
| High Level Macro | do_colon( ) | Points to an array of macro addresses. |

The corresponding compiler function are:

| INTERPRETER | COMPILER |
|---|---|
| do_variable( ) | Compile_variable( ) |
| do_constant( ) | Compile_constant( ) |
| do_colon | Compile_colon( ) |

Given that the address of the dictionary header for a macro is in WA, its Code Field Address is executed by:

```
(*WA->CFA)();
```

WA is a pointer to a DictHeader structure. CFA is a field within that structure. C++ executes the function within the first parentheses (*WA->CFA), which is a pointer to a function. The second parenthesis

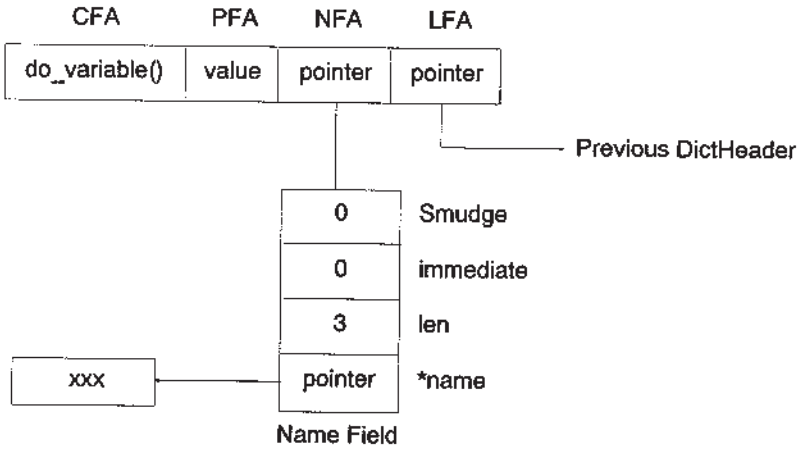
pair indicates there are no C++ arguments. Remember that all CALC arguments are passed on the Data Stack, not the internal C++ stack.

**VARIABLE EXECUTION** Referencing a variable name pushes its address onto the Data Stack. The Outer Interpreter parses a word, then performs a dictionary search. WA is set to the DictHeader address of the variable. do_variable( ) pushes the address of the PFA onto the Data Stack.

```
void do_variable()
{
     pushsp((long) &WA->PFA.lvalue);
}
```

The definition of variable xxx and its DictHeader entry are:

```
var xxx
```



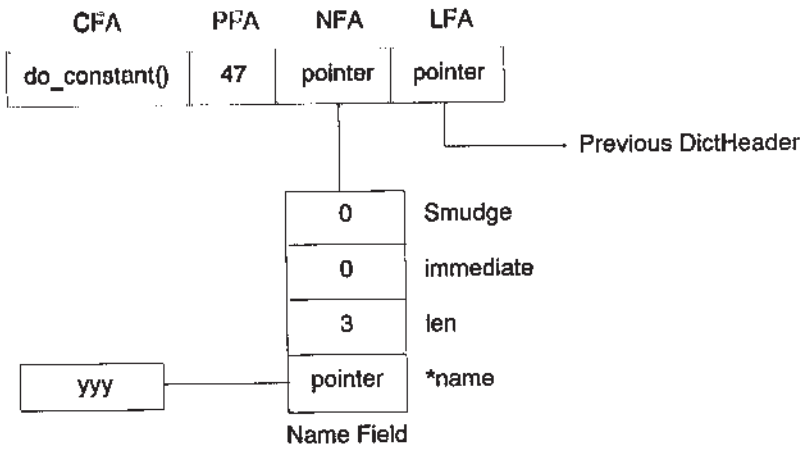

**CONSTANT EXECUTION** Using a constant pushes the value of the constant onto the Data Stack. WA for the constant is set by the dictionary search. Executing do_constant( ) pushes the contents of the PFA onto the stack.

```
void do_constant()
{
     pushsp(WA->PFA.lvalue);
}
```

The definition of constant yyy and its DictHeader entry is:

```
47 const yyy
```






**HIGH LEVEL MACRO EXECUTION** Executing a high level macro is the most complicated of the macro classes. A high level macro is a macro defined using m{ ... }m. The PFA is a list of DictHeader addresses. The CFA contains the address of the primitive C++ function do_colon( ). The code for do_colon is:

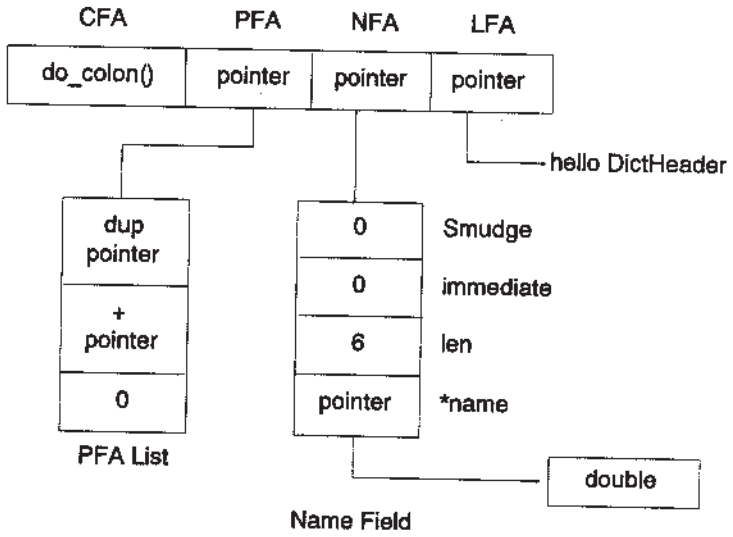
```
void do_colon()
{
     register DictHeader *word;
     DictHeader **old_IP;

     old_IP = IP;
     IP     = WA->PFA.Waddr;
     WA     = 0;
     word   = *IP++;

     while(word){
          WA   = word;
          (*word->CFA)();
          word = *IP++;
     }
     IP = old_IP;
}
```

Global variables WA and IP are used in addition to local variables. Nested high level macros recursively call do_colon( ) so the local save pointer works well and is very simple.

Type the definition of macro double:

```
m{ double                    { n -- n*2 }
     dup +
     }m
```

Its DictHeader entry is:




The pointers in the PFA List are pointers to the DictHeader entries for the macros to execute.

The Link Field Address (LFA) points to the DictHeader entry for the last macro defined. This example assumes no other macros have been defined since starting CALC. Therefore, the LFA for double points to the macro hello.

Type 2 **double** then press **Enter**:

```
CALC> 2 double
4
```

The interpreter performs the following steps to process the line:

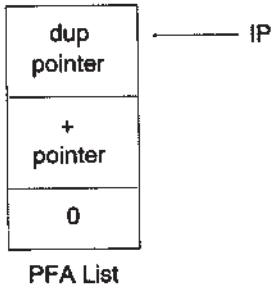
1. Parse the first word (2), convert it to a number, and push it onto the Data Stack.
2. Parse "double."
3. Search the dictionary for a macro named double.
4. Set WA to the address of the DictHeader for double.



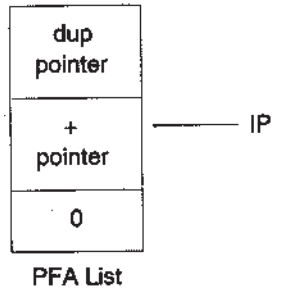
5. Call the function in the CFA, do_colon( ), to execute double.
6. Return to the Outer Interpreter.

We need to look at the steps do_colon( ) performs to run double to understand all of the processing going on to execute a high level macro definition:

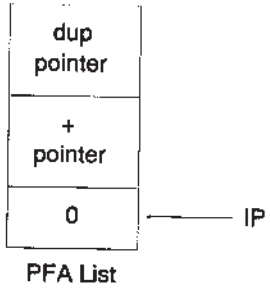
1. Save the current IP so it can be restored at the end of the macro execution.
2. Set IP to the PFA List pointed to by the PFA.




3. Call the function pointed to by IP. dup is called to duplicate the 2 on the data stack.
4. Increment IP.




5. Call the function pointed to by IP. + adds the top two numbers on the Data Stack, then pushes the sum onto the Data Stack.
6. Increment IP.




7. The IP points to 0, indicating the end of the PFA List. The do_colon( ) loop ends at this point.
8. IP is restored to the original value.
9. do_colon( ) returns to the calling function, outer( ) in this case.

Saving and restoring IP on entry and exit allows nesting of macros. That is, a high level macro can call another high level macro. do_colon( ) is called recursively in nested macros.

## PARSING THE INPUT STREAM

The final major operation performed by the Outer Interpreter is parsing words from the input line to execute. The C++ function word( ) performs this operation.

word( ) expects the delimiter character, a blank for the interpreter, on the top of the Data Stack. It reads up to that character or end of line and extracts characters into PAD. The code is:

```
void word()
{
     char ch;
     char delim;
     int  i;
     long X;

     if(IN  >= tib_len){
          read_next_line();
     }
     X       = popsp();
     delim   = (char) X;
     ch      = tib[IN++];

     while(ch == delim){
          ch = tib[IN++];
          if(IN > tib_len){
               ch = delim;
               read_next_line();
          }
     }
     i = 1;
     while(ch != delim){
          pad[i++] = ch;
          ch       = tib[IN++];
          if(IN > tib_len){
               break;
          }
     }
     pad[i++] = NULL;
     pad[i]   = delim;
     pad[0]   = i-2;
     pushsp((long)pad);
}
```

The first while loop skips leading delimiters. This handles leading blanks if the source is indented. The second while loop copies characters into pad until the next delimiter is found. The global variable IN keeps track of the current position in tib from one execution to the next. word( ) is not available to the user in CALC but easily could be made a primitive macro.

## SUMMARY

A TIL is a unique program interpreter. It contains not one, but two built-in interpreters. TILs do not follow other interpreters, such as BASIC, which may interpret the source code every time a command is executed. Until compiles source code into dictionary entries and only has to execute function addresses. Thus Until is both compiled and interpreted at the same time.

The next chapter describes the CALL compiler. A thorough understanding of the compiler is not necessary to effectively use CALC or develop your own application language with the tools in this book.

# Chapter 9

# THE CALL COMPILER

## INTRODUCTION

This chapter describes the Callable Application Language Library (CALL) compiler and explains its operation. CALL is the RPN-style compiler used by Until. Before getting into the detailed discussions, I want to point out that I probably would not have finished the compiler without the excellent debugging features built into Borland C++.

Simplicity is an overriding tenet in both CALL and Until. Simplicity makes the software more understandable for the average C/C++ programmer. The code is written with clarity as a prime concern. Many code optimizations that are possible are not included for this reason.

**WHY AN RPN COMPILER?** The first answer is simplicity. A language based on RPN is very simple to parse, especially compared to a traditional programming language like C or Pascal. Nested commands are not necessary. In fact, parentheses are not available in most RPN systems. HP calculators don't even have '(' and ')' keys.

An adjustment period may be necessary before becoming comfortable with RPN syntax. Once you do, RPN seems as natural as the algebraic system that takes years to teach in elementary and high school. The main thing is to give RPN a chance before discarding it.

The simplicity of RPN is very attractive from the standpoint of writing a compiler. Procedural language compilers are complex beasts, with symbol tables, that must parse nested operations and require complex code generation and optimization phases.

A compiler for an RPN-based language, in contrast, requires no separate symbol table, can use very simple parsing methods, and do not have to build stack frames for every procedure call.

The final reason for my choosing an RPN compiler is familiarity. I have used TILs for several years. I understand how they work. Developing a compiler is much easier when you understand how it works.

## CALL COMPILER OVERVIEW

The CALL compiler does not compile directly to machine language; it compiles to a list of function addresses for the interpreter to execute. It is possible for a TIL to compile directly to machine language. Several commercial Forth implementations compile to machine language.

There are two kinds of addresses manipulated by the compiler. These are the CFA and PFA List. The address stored in a macro's CFA must be the address of a primitive C++ function. The PFA List is a list of *DictHeader* addresses of macros to execute. The compiler is modularized with a function to build a skeleton DictHeader structure and functions to compile each class of operator. The operator classes are:

- Variables
- Constants
- Primitives
- High Level Macros
- Literals
- Loops and Branching
- Conditionals

The operation of each class is discussed in the following sections. Many of the compilation macros are defined as immediate macros. An immediate macro executes "immediately," even during compilation, rather than being compiled into the macro definition. All of the immediate compilation macros execute separate compile and execution time C++ primitives. For example, Compile_loop( ) is the compile time loop function and do_loop( ) is the execution time function for the ( ... ) loop operators.

**VARIABLES** One of the simpler compiling macros is var. An example of compiling a variable is:

```
var xxx
```



var executes Compile_variable( ) to compile a variable. It performs the following steps to compile the variable xxx:

1. Parse the next word from the input stream (xxx).
2. Create a skeleton DictHeader structure named xxx.
3. Set the CFA to do_variable( ).

The code for Compile_variable( ) is:

```
void Compile_variable()
{
     next_word();
     Create();
     DP->NFA->smudge     = 0;
     DP->NFA->immediate  = 0;
     DP->CFA             = do_variable;
     DP->PFA.lvalue      = 0;
}
```

See Chapter 8 for a sample DictHeader for a variable.

**CONSTANTS** Compiling constants is similar to compiling variables. Defining a constant such as yyy is:

```
47 const yyy
```

yyy is initialized with a value of 47. The steps to compile the constant are:

1. Parse the next word from the input stream (yyy).
2. Create a skeleton DictHeader structure named yyy.
3. Set the CFA to do_constant( ).
4. Load the PFA with the value on the top of the Data Stack (47 for yyy).

The code for Compile_constant( ) is:

```
void Compile_constant()
{
     next_word();
     Create();
     DP->NFA->smudge     = 0;
     DP->NFA->immediate  = 0;
     DP->CFA             = do_constant;
     DP->PFA.lvalue      = popsp();
}
```

See Chapter 8 for a sample constant DictHeader.

## PRIMITIVES

Compiling a primitive macro is not done directly by the compiler. The C++ functions build_primitive( ) and build_iprim( ) set up primitive macros. All of the calls to these two functions are isolated in make_prims( ), which is called as part of startup processing.

Macros must exist before they are referenced in CALC. Primitive macros are really C++ functions. This combination makes direct compilation impractical as a macro must exist before it can be referenced. build_primitive( ) and build_iprim( ) are discussed in Chapter 10.

## HIGH LEVEL MACROS

The single most complex part of the compiler is compiling high level macros. Three main functions do the bulk of the work:

- Create( )
- Compile_colon( )
- build_pfa_list( )

Create( ) is called by all of the compiling macros to create an empty macro definition in the dictionary. Compile_colon( ) is the main compiler function. build_pfa_list( ) adds address tokens to the PFA List data structure, which is used to create the final macro PFA. The code for Compile_colon( ) follows:

```
void Compile_colon()
{
     void *ptr;
     int  len;

     if(STATE == 1){
          cout << "ERROR->Already in Compile
Mode.\n";
     }
     next_word();
     Create();
     STATE      = 1;
     pfa_offset = 0;
     build_pfa_list();
     len        = popsp();
     len       += 4;
     ptr        = new DictHeader**
[len*sizeof(DictHeader*)];
     memcpy(ptr,pfa_list,(len*sizeof(DictHeader*)));
     DP->NFA->smudge    = 0;
     DP->NFA->immediate = 0;
     DP->CFA            = do_colon;
     DP->PFA.Waddr      = (DictHeader**)ptr;
}
```



The compiler is invoked by the primitive *m{*. The first step is checking the current compile STATE. Nested macro definitions are not allowed. STATE is 0 during normal interpretation and 1 during compilation. next_word( ) parses the next word from the input string. The parsed word is the macro name to create. It is in the character array pad. Create( ) is called to create a skeleton macro header in the dictionary.

The PFA List is then constructed by build_pfa_list( ). A fixed length work structure is used, then space sized to fit the macro definition is allocated using the C++ new operator, and the actual size used is copied to the final Parameter Field. The final step is finishing the macro definition.

The smudge flag is set in Create( ) to prevent accidental matches in dictionary searches done before the compile is completed. One of the last steps in Compile_colon( ) is clearing the smudge flag to make the macro name visible during later dictionary searches.

build_pfa_list( ) is a second major function in the compilation process. The code is:

```
void build_pfa_list()
{
     int  status;
     char *string;
     DictHeader *save_WA;

     save_WA    = WA;
     next_word();
     while(STATE){
          status = prim_find();
          switch(status){
          case -1:
               (WA->CFA)();
               if(STATE == 0){
                    goto FinishDef;
                    }
               break;
          case 0:
               pushsp((long)pad);
               number();
               if(ABORT){
                    string = &pad[1];
                    cout << string << ": Word Not
                                          Found\n";
                    STATE = 0;
```

```
                    abort_F();
                    break;
               }else{
                    literal();
               }
          case 1:
               pfa_list[pfa_offset++] = WA;
               break;
          default:
                    cout << "Bad prim_find
                         return-build_pfa_list\n";
          }
          if(pfa_offset > PFA_SIZE){
               cout << "Definition too long. \n";
               abort_F();
               STATE = 0;
          }
          next_word();
     }
FinishDef:
     pfa_list[pfa_offset] = 0;
     WA                   = save_WA;
     pushsp(pfa_offset);
}
```

build_pfa_list( ) is a big loop that:

- Parses the next word from the input stream
- Looks it up in the dictionary
- Executes the macro if it is immediate
- Compiles the macro into the PFA List if it is a normal macro
- Converts numbers into literals and compiles the number into the PFA List
- Aborts if the word is undefined

The loop continues as long as compile mode is active (STATE = 1). The end macro operator, }m, resets STATE to 0, which changes the compile mode back to interpret. WA is saved and restored to keep the previous thread in tact. The other classes of operators in the compiler are implemented as immediate words that execute in compile mode to build entries in the PFA List.

## LITERALS

Literals pose minor problems. Typing a number in interpretive mode simply pushes the number onto the Data Stack. That will not work during compilation. The numeric literal must be compiled into the dictionary.



The C++ function, *literal( )*, compiles a literal into a macro definition. At run time, the primitive *do_lit( )* extracts the literal from the dictionary and pushes it onto the stack. The steps literal( ) performs are:

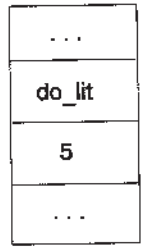
1. Compile the address of do_lit( ) into the dictionary.
2. Pop the value on the top of the Data Stack and compile it into the next location in the dictionary.

The code for literal is:

```
void literal()
{
     pfa_list[pfa_offset++] = LIT_WA;
     WA                     = (DictHeader*)popsp();
}
```

Notice the literal placed in WA is type cast to a DictHeader pointer. WA is compiled into the dictionary by build_pfa_list( ). LIT_WA contains the address of do_lit( ). The PFA List is defined as an array of DictHeader pointers.

The following is a PFA List segment when a '5' is compiled in a macro definition:




The run-time CFA contains do_lit( ). The code for do_lit( ) is:

```
void do_lit()
{
     pushsp((long) *IP++);
}
```

The literal is type cast back into a long integer as it is extracted from the dictionary. IP is manipulated so the interpreter never sees the literal.

## BRANCHING, LOOPS, AND CONDITIONALS

Looping and branching go hand in hand. CALL does not allow user specified branching, i.e., there is no GOTO operator. Branching is still necessary for both loops and conditionals. This section discusses first branching, then the loop operator.

**BRANCHING** Only two branching primitives are necessary. These are:

- branch( )
- zero_bran( )

Both branches are implemented as relative jumps. branch( ) is an unconditional branch. It is used at the end of a loop to branch back to the beginning. zero_bran( ) is a conditional branch. The branch is taken if the value on the top of the Data Stack is zero. The compile-time processing of calculating the branch offset is performed by the loop and conditional primitives. The code for branch( ) is:

```
void branch()
{
     long offset;

     offset = (long) *IP;
     IP    += offset;
}
```

The Outer Interpreter uses IP as its Instruction Pointer. Simply modifying IP has the effect of performing a jump in the code. The offset can be either positive or negative.

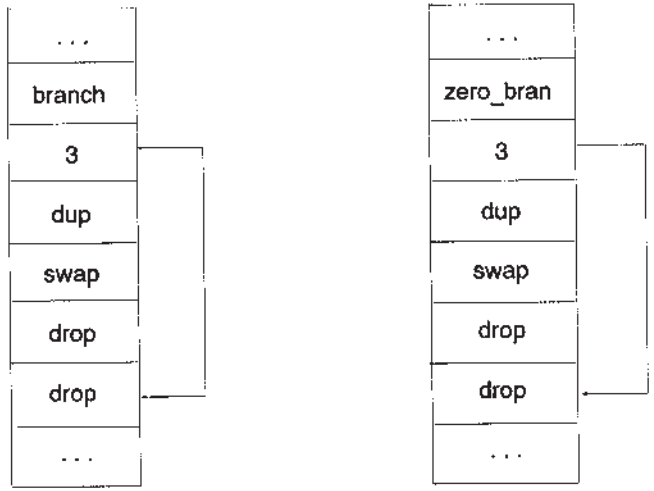
The code for zero_bran( ) is:

```
void zero_bran()
{
     long truth;
     long offset;
          truth = popsp();
     if(truth){
          IP++;          // Skip branch offset
          return;
     }
     offset = (long)*IP;
     IP    += offset;
}
```

zero_bran( ) takes the branch if the Data Stack entry is zero. If the value is nonzero, the branch is not taken. The IP must be incremented to skip over the branch offset in this case.



The following PFA List segments illustrate entries for a branch and zero_bran when the branch is taken:




**LOOPING** CALL supports only one looping construct, ( ... ). Other structured looping constructs, such as *begin ... until,* can be easily added to the compiler. Looping operators are restricted due to space limitations and to keep CALC simple for the user to learn.

The macro '(' invokes Compile_do and ')' calls Compile_loop. Compile_do( ) is deceptively simple, performing the following steps:

- Compile the run-time do function address into the PFA list
- Save the address of the empty branch offset on the Return Stack

Compile_loop( ) is slightly more complex. It does the following steps at compile time:

- Compile the run-time function for loop
- Retrieve the address of the branch offset cell from the Return Stack
- Calculate the branch offset
- Compile the branch offset into the PFA List

The code for Compile_do( ) and Compile_loop( ) follows:

```
void Compile_do()
{
```

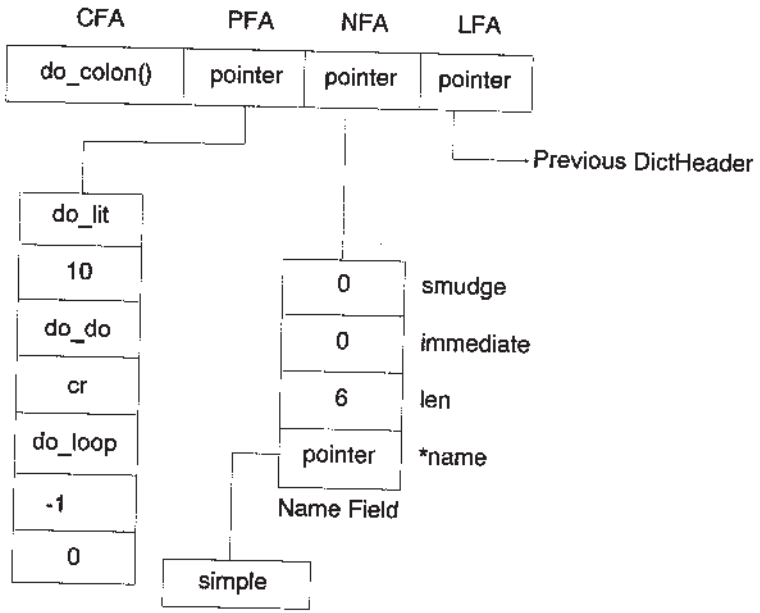
```
        pfa_list[pfa_offset] = DO_WA;
        pushrs((long)&pfa_list[pfa_offset++]);
}

void Compile_loop()
{
        long        offset;

        pfa_list[pfa_offset++] = LOOP_WA;
        r_from();
        offset  = popsp();
        offset  = offset - (long) &pfa_list[pfa_offset];
        offset  = (offset / sizeof(DictHeader*)) + 1;
        pfa_list[pfa_offset++] = (DictHeader*) offset;
        WA      = 0;
}
```

The do loop is valid only in compile mode because it manipulates the PFA List.

Define a macro that does a simple loop:

```
m{ simple                    { -- }
   10 ( cr )
}m
```

*simple* prints 10 blank lines on the screen. The following dictionary entry is generated for *simple*:



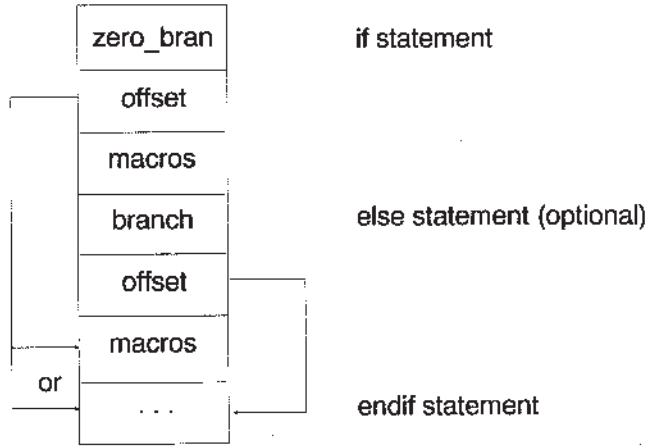



The run time function called by '(' is do_do( ). It pushes the number of times to loop and a zero for the index onto the return stack. The run-time function called by ')' is do_loop( ). do_loop( ) calls branch( ) to jump to the top of the loop. Each time do_loop( ) executes, the loop index on the Return Stack is incremented and tested to see whether to loop again or fall out of the loop.

Other structured looping constructs, such as *begin ... again*, *do ... while*, and *begin ... until*, are easily implemented in CALL. Each is a variation on the ( ... ) loop presented here. Another useful enhancement to the looping constructs is a function to explicitly exit a loop.

**CONDITIONALS** *if ... else ... endif* is the only conditional statement in CALL. if executes the code following the if or branches depending on the truth value on the top of the Data Stack. Compile_if( ), Compile_else( ), and Compile_endif( ) are the compile-time primitives that handle conditional compilation.

The following diagram shows the internal branching logic to handle an if ... else ... endif statement:




The if statement generates a zero branch to either the point of the else or the endif statement. An unconditional branch to the endif statement is generated where the else occurs in the code.

**if** The if statement is relatively simple. It compiles the address of zero_bran( ) into the PFA List, allocates an empty PFA List cell, and pushes the address of the empty cell onto the Return Stack. The address

is used later by else or endif to calculate the relative offset for zero_bran( ). The code for Compile_if( ) is:

```
void Compile_if()
{
     pfa_list[pfa_offset++] = ZERO_BRAN_WA;
     pushrs((long)&pfa_list[pfa_offset++]);
     WA = 0;
}
```

**endif** An if is bounded by endif. The compile-time action for endif is performed by Compile_endif( ). The steps this function performs are:

- Calculate the offset for zero_bran( ) in the if
- Compile the offset into the PFA List at the if

There is no run-time code associated with an endif. The code for Compile_endif( ) is:

```
void Compile_endif()
{
     DictHeader **branch;
     long       offset;

     r_from();
     offset  = popsp();
     branch  = (DictHeader**) offset;
     offset  = (long) &pfa_list[pfa_offset] - offset;
     offset  = offset / sizeof(DictHeader*);
     *branch = (DictHeader*) offset;
     WA      = 0;
}
```

**else** An else statement has two primary purposes: fill in the offset for the previous if and set up a branch to skip the code between else and endif. The else code is basically the code for if and endif combined with a few minor changes. The steps that compile the else portion of an if ... else ... endif statement are:

- Compile the address of branch( ) into the PFA List
- Allocate an empty PFA List cell
- Push the address of the empty cell onto the Return Stack
- Calculate the branch offset for the if
- Compile the branch offset for the if

The code for Compile_else( ) is:

```
void Compile_else()
{
     DictHeader **branch;
```



```
     long       offset;

     r_from();
     pfa_list[pfa_offset++] = BRANCH_WA;
     pushrs((long)&pfa_list[pfa_offset++]);
     offset  = popsp();
     branch  = (DictHeader**) offset;
     offset  = (long) &pfa_list[pfa_offset] - offset;
     offset  = offset / sizeof(DictHeader*);
     *branch = (DictHeader*) offset;
     WA = 0;
}
```

## SUMMARY

This chapter described CALL. It handles several classes of compilation macros including variables, constants, macros, literals, loops, and conditionals. Several additional looping constructs can easily be added, such as begin ... until or do ... while, with little effort.

The CALL compiler defines the application language. An RPN-based compiler is much simpler than a C or Pascal compiler. The compiler resolves macros to C++ function addresses. The flavor of your application language can be changed by replacing this compiler with a compiler of your own.

# Chapter 10

# UNTIL CORE PRIMITIVES

## INTRODUCTION

The lowest level Until/CALL primitives are C++ functions. This chapter discusses these functions and how to add primitives of your own. Primitive C++ functions are typically not referenced or run directly by the user.

## ADDING PRIMITIVES

Previous sections have described many of the primitive operators in Until. Adding new primitive operators to Until is easy. One of the primary aims of Until is easily adding new primitive functions. The ability to easily add new primitives is an important consideration in extending the base system and keeping execution efficiency high. The steps to add a new Until primitive is slightly different than adding a CALL primitive. The following section addresses adding new primitives to each.

**ADDING AN UNTIL PRIMITIVE** In simplest terms, write a C++ function and add an entry for it in make_prims( ). There is only one rule for interfacing the new function with the rest of Until; all parameters used by the function must be passed via the Data Stack. The primitive functions pushsp( ) and popsp( ) already exist for referencing the Data Stack.

The process of adding a new primitive is illustrated by the following example. A common CALC program sequence is:

```
1 -
```

to decrement the number on the top of the Data Stack. A program that uses the "1 -" sequence a lot can be speeded up by adding a new primitive that subtracts 1 from the top stack entry.

Step 1 is writing a new primitive function:

```
void one_minus()
{
     long value;

     value = popsp();     // Extract top of stack number
     pushsp(value - 1);   // Decrement and it put back
}
```

The function popsp( ) retrieves the top value from the Data Stack. pushsp( ) is its inverse; pushsp( ) pushes a long int onto the Data Stack.

Step 2: add the following line to make_prims( ) in USER.CPP:

```
build_primitive("1-",one_minus);
```

The first argument to build_primitive( ) is the macro name string. The second argument is the address of a C++ function to execute. So when the user types "1–" the function one_minus( ) is executed by the interpreter.

The following steps are also needed to compile the new function into Until:

1. Add the function prototype to UNTIL.H.
2. Add the source filename that contains the new primitive to the project file (CALC.PRJ).
3. Compile and link CALC.

At this point, a new primitive macro named 1– has been added to CALC and Until. Follow these steps to add any new primitive macro.

**ADDING CALL PRIMITIVES** Adding a new primitive to CALL is a bit more complicated than adding one to Until. The Until primitive is set up to be compiled into the dictionary by CALL, then executed by the outer interpreter. A CALL primitive on the other hand has both a compile time action and a separate run-time action. All compile-time CALL primitives are immediate.

The general steps for adding a new CALL primitive are:

- Create a new compile-time primitive
- Create a run-time primitive
- Add the appropriate entries to make_prims( )



New compiler primitives generally manipulate the pfa_list data structure.

The example used for this discussion is adding a new looping construct. The only loop defined in CALC is ( ... ). ( expects the number of times to execute on the Data Stack. A very simple extension is **do_for ... loop**. Much of the code necessary to implement the do_for construct already exists in the ( operator:

| MACRO | COMPILE-TIME FUNCTION | RUN-TIME FUNCTION |
|---|---|---|
| ( | Compile_do | do_do |
| ) | Compile_loop | do_loop |
| do_for | Compile_do_for | do_for |
| loop | Compile_loop | do_loop |

The only new run-time function is do_for( ). This corresponds to the Compile_do_for( ) at compile time. do_for is a more general case of (. The detailed discussion of the code for Compile_do( ) and Compile_loop( ) is in Chapter 9. The code for the run-time functions follows.

```
void Compile_do()
{
     pfa_list[pfa_offset] = DO_WA;
     pushrs((long)&pfa_list[pfa_offset++]);
}
```

Compile_do( ) adds the function address for do_do( ) to the pfa_list and leaves its address on the Return Stack for Compile_loop( ) to use to set up the correct branch point at the end of the loop.

```
void Compile_do_for()
{
     pfa_list[pfa_offset] = DO_FOR_WA;
     pushrs((long)&pfa_list[pfa_offset++]);
}
```

The operation of Compile_do_for( ) is identical to Compile_do( ) with the exception of the address compiled into the pfa_list. It compiles the function address for do_for( ) rather than do_do( ).

```
void do_do()
{
     pushrs(0);
     one_minus();
     to_r();
}
```

The code for do_do( ) has been simplified here for this discussion. The code in the source file contains additional checks for stack underflow. do_do( ) is called when macro ( is called by the interpreter:

```
10 ( .... )
```

Two counters are necessary for a loop, the current index and the ending index. do_do( ) uses zero as the starting current index value. One is subtracted from the ending index because loop indexing is zero based. Both indexes are pushed onto the Return Stack. Thus, the loop index always counts from zero for the specified number of iterations.

```
void do_for()
{
     to_r();
     one_minus();
     to_r();
}
```

do_for( ) is identical to do_do( ) except the starting index is also picked up from the parameter stack:

```
100 90 do_for ... loop
```

The starting index is 90 in this example. Thus, when the macro i is executed in the loop, the first value will be 90 and the last value will be 99. Stack under flow tests should be added to the beginning of do_for( ) to be sure there are at least two entries on the Data Stack if you add do_for( ) to your version of CALL.

```
void do_loop()
{
     long end;
     long index;

     end   = poprs();
     index = poprs();
     if(index == end){
          IP++;
     }else{
          index++;
          pushrs(index);
          pushrs(end);
          branch();
     }
}
```

The code for do_loop( ) is identical for both types of do loop. The end of the loop is tested and the branch taken back to the start of the loop. do_loop( ) illustrates a point about macro names here. It exists under two separate macro names after do_for ... loop has been added.



The final step in adding a CALL primitive is adding entries to make_prims( ). The following entries are necessary for both do_for ... loop and ( ... ):

```
build_primitive("do_do",do_do);
build_primitive("do_for",do_for);
build_primitive("do_loop",do_loop);

build_iprim("(",Compile_do);
build_iprim(")",Compile_loop);
build_iprim("do_for",Compile_do_for);
build_iprim("loop",Compile_loop);

DO_WA          = set_WA("do_do");
DO_FOR_WA      = set_WA("do_for");
LOOP_WA        = set_WA("do_loop");
```

The reference card entries for these new macros are:

**(** — **{ n --- }**
"start loop". Begin a do loop. Can be used only in a macro definition.

```
m{ ten-dots 10 ( " ." ) }m
```

See ')'.

**)** — **{ --- }**
"end loop". The end of a do loop. Can be used only inside a macro definition. See '('.

**do_for** — **{ end start --- }**
"do for". Begin a do loop. Can be used only in a macro definition.

```
m{ ten-dots 10 ( " ." ) }m
```

See 'loop'.

**loop** — **{ --- }**
"loop". The end of a do loop. Can be used only inside a macro definition. See 'do_for'.

Adding new CALL primitives requires some thought and careful attention to detail. Adding additional looping constructs such as *begin ... until* or *do ... while* is not difficult because the existing looping constructs can be used as a template.

## PRIMITIVES

The code that makes up Until, CALL, and CALC uses a little over 2,000 lines. Most of the functions are less than half a page in length. CALC is a completely functioning language. Most conventional languages are many thousands of lines of code. The simplicity of the overall approach using a TIL and RPN syntax have a great deal to do with the small size.

Most functions/macros that users will access in CALC have already been discussed at some point in this book. The same is true for the interpreter and compiler. The remainder of this chapter identifies the lower level C++ functions not covered elsewhere. Most of the functions are very short and simple, so the source code is not included in the text. Complete source code is on the disk included with this book. The functions are divided into and discussed in several related areas.

**STACK MANIPULATION FUNCTIONS** Functions that perform basic stack manipulation are discussed here. Until uses two stacks, the Data Stack and the Return Stack. There are separate functions for each. The stack manipulation functions are:

| | |
|---|---|
| pushsp | Push a number onto the Data Stack |
| popsp | Pop a number off the Data Stack |
| drop | Executed by the drop macro |
| store | Executed by -> |
| prim_dup | Executed by the dup macro |
| swap | Executed by swap |
| rot | Executed by rot |
| rfetch | Fetch a value from the Return Stack |
| to_r | Move the top of the Data Stack on the Return Stack |
| r_from | Retrieve the top of the Return Stack to the Data Stack |
| times | Executed by * |
| divide | Executed by / |
| plus | Executed by + |
| minus | Executed by – |

**INTERPRETER FUNCTIONS** These functions are all used internally by the interpreter:

| | |
|---|---|
| word | Parse the next word from the input stream |
| minus_find | Look up string whose address is on the top of the Data Stack in the dictionary |



| | |
|---|---|
| exec_word | Execute a macro from a C++ function |
| prompt | Type the prompt on the screen |
| next_word | Parse and look up next word in the input stream |
| read_next_line | Read next line from input stream |

**COMPILER FUNCTIONS** The following functions are internal to the CALL compiler.

| | |
|---|---|
| CreateNFA | Creates the Name Field entry in the dictionary |
| prim_find | Searches dictionary for string at PAD |
| build_primitive | Creates dictionary entry for a primitive C++ function |
| build_iprim | Creates dictionary entry for primitive C++ immediate function |

**STARTUP/SHUTDOWN FUNCTIONS** The low level functions associated with CALC startup and shutdown are:

| | |
|---|---|
| cold | Perform cold start |
| bye | Return to DOS; called by exit |
| read_include | Read next line from CALC.APP |
| close_include | Close CALC.APP |
| do_autoload | Compile contents of CALC.APP |

**MISCELLANEOUS FUNCTIONS** The remaining functions do not fit into any of the above categories:

| | |
|---|---|
| counted_to_null | Convert counted string to null terminated form |
| null_to_counted | Convert counted string to null form |
| less_than | Executed by < |
| greater_than | Executed by > |
| equals | Executed by == |
| dot | Executed by = |
| words | Executed by macros |
| cr | Executed by cr |
| dot_s | Executed by s? |
| gdot_s | Executed by .s |
| parens | Executed by { |
| mod | Executed by rem |

| | |
|---|---|
| dot_quote | Executed by " |
| help | Executed by help to look up help entry for a macro |

## SUMMARY

Adding C++ primitives to an application language is central to the theme of this book. It is a very powerful tool that lets the programmer customize the feel and function of the system. The capability of easily adding new primitives is one of the unique features of Until.

Most of the macros that are primitives can be called directly from C++. That means system access goes in both directions; Until can access the application program's data structures and the application program can use the features built into Until.



# Chapter 11

# THE HELP UTILITY

## INTRODUCTION

The Until help facility is very simple but functional for many applications. This chapter looks at Until's help as a stand-alone utility. The help( ) function in Until is located in PRIM.CPP. The stand-alone program is in HELP.CPP. Both versions of help function identically. There are minor changes in HELP.CPP to eliminate having to include a lot of other Until code.

## THEORY OF OPERATION

The operation of help is simple. It reads a text file with each searchable topic, beginning in column 1. A blank terminates the topic name. When a topic match is found, that line and each subsequent line is printed until a nonblank character occurs in column 1. The default help filename is HELP.APP. A segment of HELP.APP follows to illustrate the file layout:

+ { n1 n2 --- n1+n2 }
"plus". Add n1 to n2. The sum is placed on the top of the stack.

```
1 2 +
```

The result, 3, is left on the top of the stack.

" text" { --- }
"double quote". Type a string to the screen. Used in the form:

```
" This goes to the screen."
```

help { --- }

"help". Looks up the specified macro name in HELP.APP and prints the entry to the screen. Usage:

```
help var
```

prints the help entry for "var." Use the 'macros' command to print the macros defined.

The example text contains three searchable topics; +, ", and help. Any other topic will result in a "Topic not found." message.

The bottom line is a simple help system that can be set up, either stand-alone or added into another program by using HELP.CPP and building a HELP.APP file.

## HELP CODE

HELP.CPP was created by extracting the function help( ) from PRIM.CPP. Next a simple main( ) was added to call help( ). There were several changes to help( ) to remove dependencies on other Until code. All of the I/O related code was changed to use a single file descriptor rather than an array. The call to word( ) was removed and the search topic passed to help( ) instead. All in all, the modifications required to build HELP.CPP as a stand-alone program were very minor.

The program starts off by referencing several system include files:

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
```

Next, constants and variables used by the program are declared. The FILE pointer is a single entry rather than an array because only a single file is used. The function prototype for help( ) is also in this code segment.

```
const TRUE   = 1;
const FALSE  = 0;
const BLANK  = ' ';

FILE *FD;
char pad[512];

void help(char *topic);
```



Function main( ) is very short. It is set up to take the search topic from the command line. This is accomplished by including argc and argv as parameters. argc is the number of command line arguments. The first argument, argv[0], is the name of the program being executed. The second argument, argv[1], is really the first argument used by the program. Therefore, argc is 2 when a search topic exists. The user is queried for a search topic if one is not input on the command line. The final step in main( ) is the call to help( ) with the search topic as its sole parameter.

```
void main(int argc, char*argv[])
{
     char topic[80];

     if(argc < 2){
          cout << "Enter Search Topic: ";
          cin  >> topic;
     }else{
          strcpy(topic,argv[1]);
     }
     help(topic);
}
```

The function that does all of the work is help( ). It takes a character string as its argument and searches HELP.APP for a matching topic. When a match is found, the entry is output to the screen. The first several lines define local variables used in processing.

```
void help(char *topic)
{
     int  i;
     int  j;
     int  match;
     int  truth;

     char *status;
     char line[132];
     char string[32];
```

The first processing step is opening the help file. The code assumes HELP.APP is in the current directory. Two simple enhancements are adding a specific directory to the filename or passing the filename to help( ) in addition to the search topic. An error message is displayed if HELP.APP does not exist.

```
FD = fopen("HELP.APP","r");
if(!FD){
     cout << "Help file not found\n";
     return;
}
```

The next group of lines perform some initialization steps. The truth flag is set to FALSE. truth is used as the end of processing to determine whether or not to display an error message. FALSE means the search topic was not found; print the error message. TRUE indicates the topic was found so no error message is generated to the screen. The work area, pad, is initialized with the search topic and the first line read from HELP.APP.

```
truth  = FALSE;
for(i=0;i<32;i++){
     pad[i] = 0;
}
strcpy(pad,topic);
status = fgets(line,132,FD);
```

Processing with pad is not really necessary. The code is needed in the Until version and left in the HELP.APP for code compatibility.

The main processing loop of help( ) is a while loop that continues until end of file is reached or the search topic processed. The first if statement tests the first character of the line for a blank. If the character is nonblank, the first word in the line is extracted into string.

```
while(status){
     if(line[0] != BLANK){
          j   = 0;
          i   = 0;
          while(line[i] != BLANK){
               string[j++] = line [i++];
          }
          string[j] = NULL;
```

The next step is comparing the string extracted from HELP.APP with the search topic in pad.

```
          match     = strcmp(string,pad);
```

strcmp( ) returns zero when the compare is successful. The second level if tests the comparison and outputs the first line to the screen.

```
          if(match == 0){
              truth  = TRUE;
              cout << line;
              status = fgets(line,132,FD);
              if(!status) goto All_done;
```

This while loop continues until end of file or the first line with a nonblank first character. Each time through the loop displays the buffer containing the previous line from the help file.

```
              while(line[0] == BLANK){
                   cout << line;
```



```
                   status = fgets(line,132,FD);
              }
              goto All_done;
```

This line goes with the test of match. When pad and string do not match, the next line is read and the search through the help file continues.

```
          }else{
               status = fgets(line,132,FD);
               if(!status) goto All_done;
          }
```

The code after this }else{ pairs with the first if statement. It effectively skips the lines with a blank first character while searching for the next topic line.

```
     }else{
          status = fgets(line,132,FD);
          if(!status) goto All_done;
     }
}
```

When end of file is detected in HELP.APP, processing branches to All_done. The file is closed and an error message displayed if the search topic was not found in the help file.

```
All_done:
     fclose(FD);
     if(!truth){
          cout << "Topic not found.\n";
     }
}
```

## SUMMARY

Until's help facility is both simple and generic. It can be added to application programs or used stand alone. help( ) also provides a convenient way to add online help by simply creating a HELP.APP file.

Any number of enhancements are practical. Some include:

- Index file of searchable topics
- Specify help filename
- Fancy formatting

An index file of searchable topics can be constructed that consists of the topic string and offset from the beginning of the help file for the entry. This technique can speed up searching significantly and is probably worthwhile if dealing with very large help files.

The help filename could be specified on the command line along with the search topic. This could provide the hooks for adding context sensitive help where different help files are used, depending on the context.

There is no formatting done with the present help( ) implementation. Adding a little formatting to the on-screen presentation of the help entry could enhance an application; for example, different colors for the search topic and help test.

Even though the Until help facility is simpleminded, it is quite useful for many applications.



# Chapter 12

# WHERE TO GO FROM HERE

## INTRODUCTION

The entire CALC, Until, CALL system has been described at this point. Hopefully, the potential power of the application language concept and its implementation in the form of CALC is appreciated and understood by now. This chapter presents some ideas for potential uses of and enhancements to the system as food for thought.

## USES AND ENHANCEMENTS

Until, as presented in this book, provides a framework to build application languages. However, there are many enhancements that are easy to make which are left to the reader. The process of adding enhancements will bring a deeper understanding of the mechanics used in Until. This will, in turn, stimulate new ideas for using Until in your application programs. The following sections suggest some uses for and enhancements to Until.

**GENERAL ENHANCEMENTS** The simplest enhancement is simply adding some additional math functions and use CALC as it is, either stand alone or inside of other programs. Floating point should be a relatively easy enhancement. See Chapter 10 for the discussion on adding primitives to Until.

Until was originally conceived as an implementation of Forth. A Forth can be easily implemented on the Until core. A relatively complete Forth requires 100 - 200 primitives. A Forth based on Until could be a very powerful applications language.

The following table maps many CALC macros to Forth words:

| *CALC* | *FORTH* | *COMMENT* |
|---|---|---|
| rem | mod | Calculate modulo or remainder |
| var | variable | Define a variable |
| const | constant | Define a constant |
| <- | @ | Fetch a value |
| -> | ! | Store a value |
| == | = | Equal |
| macros | words | Display list of macros |
| = | . | Display number on top of the stack |
| s? | .s | Nondestructive stack dump |
| { } | ( ) | Comment |
| m{ | : | Start macro/word definition |
| }m | ; | End macro/word definition |
| " | ." | Display the string in quotes |
| ( | 0 do | Start a do loop |
| ) | loop | End a do loop |
| if | if | If construct |
| else | else | Else portion of if construct |
| endif | then | End the if |
| + | + | Add |
| – | – | Subtract |
| * | * | Multiply |
| / | / | Divide |
| dup | dup | Duplicate top of data stack |
| swap | swap | Swap top two items on data stack |
| rot | rot | Rotate third entry to top of stack |
| drop | drop | Drop top stack entry |
| ? | ? | Display value at an address |

This correspondence between CALC macros and Forth words makes building a Forth-like application with Until relatively easy. The primitives to add can be derived from any of the Forth books listed in Appendix B.



**APPLICATION LANGUAGE USES** Providing users with the capability to modify program parameters at run time is a very useful purpose for an application language. Normally, providing such a feature requires a lot of code and effort for a relatively small amount of processing. The addition of a few lines of very simple code in the form of a primitive C++ function is all that is required to use Until for this purpose. An example could be specifying which of many fields to process in a given run or changing the number of lines on an output report.

One specific use I have in mind for an Until-based application language is in a system based upon a software finite state machine to control program flow. The finite state machine is a two-dimensional table of C function addresses. The columns represent high level functions, such as adding typesetting commands, and rows represent each field to be processed. Processing is a simple loop for each column of retrieving a function address and executing it. The contents of the state machine is loaded at compile time. An application language would allow different C function addresses to be loaded into the table at run time to customize each run without recompilation.

A modular application language, such as CALC, provides great potential for code reuse. This can cut development cost for a new program significantly. The application language only has to be thoroughly tested the first time it is written. In subsequent programs, only minimal testing is required. Macros developed in one program may also be reused, with little or no changes in most cases. Macros that take advantage of internal program data structures are the ones most likely to require modifications.

A good example of code reuse would be a system with a library of graphics functions. An application is constructed by writing C/C++ functions that call the graphics library in the correct order. Reuse of code in a graphics application is at a minimum using a conventional approach because each application is a custom stand-alone program. An application language with the graphics library functions set up as Until primitives would allow a single executable program for many applications. The applications would become Until macros that string the graphics calls together in the proper sequence.

Until provides user programmability in its current state. Providing program access to data structures in the application open up tremendous possibilities from the user point of view for customization. Some additional data types (char) and primitives are needed before best advantage can be taken of the programmability by application programs.

A completely new application language can be created by replacing CALL with a compiler of your own. This requires the most work and a complete understanding of both Until and the language being implemented.

## CONVERTING FROM C++ TO C

The code used in this book is C++. It makes limited use of the object oriented features in the language. For this reason, CALC, Until, and CALL can be converted to C with relatively little effort. The following areas would need modification:

- Change *new* calls to *malloc* or equivalent calls.
- Change comments from // to standard /* */ form.
- Change *cout* calls to *printf* or equivalent calls.
- Header files probably need modification.

Other minor changes may also be required depending on your C compiler. CALC will compile only with the compact memory model or larger with Turbo C++. One of the larger memory models may also be necessary with other compilers.

## SUMMARY

The concept of the application language is straightforward. Many commercial packages provide some form of application language. The tools, Until and CALL, presented in this book illustrate building a simple application language, CALC, that can be used in any C++ program. Potential use of this software is bounded only by your imagination.



# Appendix A

# SOURCE FILES

## INTRODUCTION

The disk included with this book contains the source code described in this book. This appendix describes the files. See the READ.ME file for last minute changes.

| | |
|---|---|
| CALC.APP | This file is loaded at startup time by Until. User level macros may be included in this file. |
| CALC.CPP | The code for running CALC as a stand-alone program. |
| CALL.CPP | The CALL compiler functions. |
| EXAMPLE.CPP | An example user developed primitive function. |
| EXAMPLE.APP | CALC macros from Chapter 3. |
| HELP.APP | The Help file. |
| HELP.CPP | Stand-alone help program. |
| IO.CPP | Internal I/O functions. |
| MATH.CPP | Primitives for math macros. |
| MORE.CPP | Stand-alone version of More. This is a sample stand-alone application program. |
| MORECALC.CPP | More and CALC combined into a single program. |
| PRIM.CPP | Misc primitive functions. |
| UNTIL.CPP | The Until Interpreter |
| UNTIL.H | Include file for all Until modules. |
| USER.CPP | Primitive macros set up are in this file. |

# Appendix B

# BIBLIOGRAPHY

## INTRODUCTION

Threaded Interpretive Languages are fascinating. The first reference I saw to TILs was the August 1980 *Byte Magazine* Forth issue. This appendix contains references to other books and sources of information about TILs in general and Forth in particular.

## BOOKS


Brodie, Leo; *Starting Forth*, Prentice-Hall, 1984.

Brodie, Leo; *Thinking Forth*, Prentice-Hall, 1984.

Katzan, Harry; *Invitation to Forth*, Petrocelli Books, 1981.

Kelly, Mahlon and Spies, Nicholas; *Forth: A Text and Reference*, Prentice-Hall, 1986.

Loeliger, R. G.; *Threaded Interpretive Languages*, Byte Books, 1981.

Pountain, Dick; *Object-Oriented Forth*, Academic Press, 1987.

## ORGANIZATIONS

The Forth Institute
70 Elmwood Avenue
Rochester, New York 14611

The Forth Institute publishes *The Journal of Forth Applications and Research* and sponsors the annual Rochester Forth Conference at the University of Rochester. The conference is usually held in early June.

Forth Interst Group
P. O. Box 8231
San Jose, Calif. 95155

FIG exists to promote Forth and serve as a central point for information about Forth. Membership dues are $30.00 per year, which includes a subscription to *Forth Dimensions Magazine*.



# Appendix C

# CALC REFERENCE CARD

## INTRODUCTION

This appendix contains a quick reference of the user level macros defined in CALC. These macros can be used in other macros. Most can be executed individually from the command line.

Each macro entry has the macro name and the stack comments on the first line. This is followed by a description of the word including the pronunciation in quotes.

| MACRO | DESCRIPTION |
|---|---|
| + | **{ n1 n2 --- n1+n2 }**<br>"plus". Add n1 to n2. The sum is placed on the top of the stack.<br>`1 2 +`<br>The result, 3, is left on the top of the stack. |
| - | **{ n1 n2 --- n1-n2 }**<br>"minus". Subtract n2 from n1 and place the result on the top of the stack.<br>`2 1 -`<br>The result, 1, is left on the stack. |
| * | **{ n1 n2 --- n1*n2 }**<br>"star". Multiply n1 times n2.<br>`2 2 *`<br>The result, 4, is placed on the top of the stack. |

**/** **{ n1 n2 --- n1/n2 }**
"slash". Divide n1 by n2.

```
4 2 /
```

Leaves 2 on the top of the stack.

**?** **{ address --- }**
"query". Type the contents of the address that is on the top of the stack.

```
trash ?
```

types the value stored in the variable trash. This is logically equivalent to "<- =".

**<-** **{ address --- value }**
"fetch". Fetch the value from address and place on the top of the stack.

```
trash <-
```

retrieves the value stored at trash and places it on the top of the stack.

**->** **{ value address --- }**
"store". Store the value into address.

```
5 trash ->
```

moves 5 into the variable trash.

**<** **{ n1 n2 --- truth }**
"less than". Compare the two numbers on the top of the stack and return TRUE or FALSE.

```
5 6 <          { returns true  }
5 5 <          { returns false }
6 5 <          { returns false }
```

**==** **{ n1 n2 --- truth }**
"equals". Compare n1 and n2. Push TRUE on the Data Stack if n1 equals n2. Otherwise, push FALSE.

```
5 6 ==         { returns false }
5 5 ==         { returns true  }
6 5 ==         { returns false }
```



**>** **{ n1 n2 --- truth }**
"greater than". Compare n1 and n2. Push TRUE onto the Data Stack if n1 > n2. Otherwise, push FALSE.

```
5 6 >          { returns false }
5 5 >          { returns false }
6 5 >          { returns true  }
```

**=** **{ value --- }**
"is" or "display". Display the number at the top of the Data Stack to the screen.

**{** **{ --- }**
"left brace". Text between the braces is treated as a comment. Left brace starts a comment.

```
{ This is a comment }
```

See '}'.

**}** **{ --- }**
"right brace". Close a comment. See '{'.

**}m** **{ --- }**
"end macro". Finishes a macro definition and exits compile mode. Must be paired with m{. See 'm{'.

**" text"** **{ --- }**
"double quote". Type a string to the screen. Used in the form:

```
" This goes to the screen."
```

The space after the first double quote is necessary.

**(** **{ n --- }**
"start loop". Begin a do loop. Can be used only in a macro definition.

```
m{ ten-dots 10 ( " ." ) }m
```

See ')'.

**)** **{ --- }**
"end loop". The end of a do loop. Can be used only inside a macro definition. See '('.

**cls** **{ --- }**
"c-l-s". Clear screen.

**const** **{ n --- }** **Compile time**
**{ --- n }** **Run time**
"constant". Create a constant.

```
5 const five
```

**cr** **{ --- }**
"c-r". Type a carriage return to the screen.

**drop** **{ n1 --- }**
"drop". Remove the top number on the stack.

**dup** **{ n1 --- n1 n1 }**
"dup". Duplicate the top stack entry:

```
1 dup
```

Results in two 1s on the stack.

**else** **{ --- }**
"else". See 'if'.

**endif** **{ --- }**
"endif". See 'if'.

**exit** **{ --- }**
"exit". Exit CALC and return to the calling function. Returns to DOS if stand-alone program.

**help** **{ --- }**
"help". Looks up the specified macro name in HELP.APP and prints the entry to the screen. Usage:

```
help var
```

prints the help entry for "var". Use the "macros" command to print the macros defined.

**i** **{ --- n }**
"i". Push the current do loop index onto the Data Stack.

**if else endif** **{ truth --- }**
"if". Tests the top of the Data Stack. When it is zero, the code after the else or endif is executed. When the value is nonzero, the code between the if and else or endif is executed.



```
5 5 ==
if    " True comparison"
else " False comparison"
endif
```

**m{** **{ --- }**
"macro start". Start a new macro definition. The macro name is the next word in the input stream:

```
m{ new-macro  }m
```

See '}m'.

**macros** **{ --- }**
"macros". Display all of the currently defined macro names on the computer screen.

**rem** **{ n1 n2 --- remainder(n1/n2)}**
"rem". Calculate the remainder of n1 divided by n2:

```
4 2 rem
```

Leaves 0 on the stack.

**s?** **{ --- }**
"stack query". Nondestructive Data Stack display to the screen.

**swap** **{ n1 n2 --- n2 n1 }**
"swap". Swap the top two entries on the top of the stack.

```
1 2 swap
```

results in the top stack entry being 2 and the second entry being 1.

**var** **{ --- }** **Compile time**
**{ --- addr.of.var }** **Run time**
"var". At compile time, creates a variable:

```
var trash
```

At run time, push the address of the variable onto the stack:

```
trash    { --- addr.of.trash }
```

# Index