Forth Interpreter User Manual

Introduction

This is a simple Forth-like stack-based interpreter implemented in standard C. It provides an educational environment for learning about stack-based computation, dictionary-based execution, and threaded interpretation.

What is Forth?

Forth is a stack-based programming language and environment known for its simplicity, extensibility, and efficiency. This interpreter implements a subset of Forth features suitable for learning and experimentation.

Features

- **Stack-based architecture**: All operations work with a data stack
- **Dictionary system**: Built-in and user-defined words stored in a dictionary
- **REPL interface**: Interactive Read-Eval-Print Loop
- **User-defined words**: Create custom functions using `:` and `;`
- **Control flow**: if-then-else, loops (begin-until, begin-while-repeat, do-loop)
- **Memory operations**: Store and fetch values from memory
- **Arithmetic and logical operations**: Complete set of mathematical and comparison operations

Getting Started

Prerequisites

- GCC compiler installed
- Standard C library (included with GCC)

Building the Interpreter

```
gcc forth.c -o forth
```

Running the Interpreter

Interactive mode:

```
./forth
```

Batch mode with a script file:

```
./forth < test.forth
```

Exiting the Interpreter

Type `quit` in the REPL to exit.

Basic Concepts

The Stack

Forth uses a Last-In-First-Out (LIFO) data stack for all operations. Numbers and intermediate results are pushed onto and popped from the stack.

Stack notation: In Forth documentation, stack effects are shown as `( before -- after )` where:

- Items before `--` are consumed from the stack
- Items after `--` are produced on the stack

Example: `+ ( a b -- sum )` means `+` consumes two numbers and produces their sum.

The Dictionary

The dictionary stores all words (functions) known to the interpreter:

- **Built-in words**: Predefined operations like `+`, `dup`, `.`
- **User-defined words**: Custom functions created with `:` and `;`

REPL (Read-Eval-Print Loop)

The interpreter runs in an interactive loop:

1. Reads input from the user
2. Evaluates each word/token

3. Prints results or errors
4. Repeats

Execution Modes

- **Interpret mode** (default): Words are executed immediately
- **Compile mode**: Words are compiled into new definitions (entered with `:`)

Built-in Words

Arithmetic Operations

Word	Stack Effect	Description
`+`	`( a b -- sum )`	Add two numbers
`-`	`( a b -- difference )`	Subtract b from a
`*`	`( a b -- product )`	Multiply two numbers
`/`	`( a b -- quotient )`	Divide a by b (integer division)
`mod`	`( a b -- remainder )`	Modulo operation

Stack Manipulation

Word	Stack Effect	Description
`dup`	`( a -- a a )`	Duplicate top stack item
`drop`	`( a -- )`	Remove top stack item
`swap`	`( a b -- b a )`	Swap top two stack items
`over`	`( a b -- a b a )`	Copy second item to top
`rot`	`( a b c -- b c a )`	Rotate top three items
`nip`	`( a b -- b )`	Remove second item
`tuck`	`( a b -- b a b )`	Insert copy of top under second

Comparison Operations

Word	Stack Effect	Description
`=`	`( a b -- flag )`	True if a equals b
`<`	`( a b -- flag )`	True if a < b
`>`	`( a b -- flag )`	True if a > b
`<=`	`( a b -- flag )`	True if a <= b
`>=`	`( a b -- flag )`	True if a >= b
`<>`	`( a b -- flag )`	True if a != b

Note: Comparison operations return -1 for true, 0 for false.

Logical Operations

Word	Stack Effect	Description
`and`	`( a b -- result )`	Bitwise AND
`or`	`( a b -- result )`	Bitwise OR
`not`	`( a -- result )`	Bitwise NOT

Memory Operations

Word	Stack Effect	Description
`!`	`( value addr -- )`	Store value at address
`@`	`( addr -- value )`	Fetch value from address

I/O Operations

Word	Stack Effect	Description
`.`	`( n -- )`	Print top of stack
`.s`	`( -- )`	Print entire stack
`cr`	`( -- )`	Print newline
`."`	`( -- )`	Print string literal

Control Flow

Conditional Execution

```
if ... then
if ... else ... then
```

Example:

```
5 3 > if ." Greater" then
10 5 < if ." Less" else ." Greater or equal" then
```

Loops

Begin-Until Loop:

```
begin ... condition until
```

Executes the body until the condition is true.

Example:

```
0 begin dup . 1 + dup 10 = until drop
```

Begin-While-Repeat Loop:

```
begin condition while ... repeat
```

Executes while condition is true.

Example:

```
10 begin dup 0 > while dup . 1 - repeat drop
```

Do-Loop:

```
limit start do ... loop
```

Executes from start to limit-1.

Example:

```
10 0 do i . loop
```

Defining Words

Word	Description
`:`	Start word definition
`;`	End word definition
`VARIABLE`	Create a variable
`CONSTANT`	Create a constant
`CREATE`	Create an expandable word

User-Defined Words

Create custom words using colon definitions:

```
: word-name ... ;
```

Example:

```
: square dup * ;
5 square .  \ Prints 25
```

Words can call other words and use control flow:

```
: factorial
    dup 1 > if
        dup 1 - factorial *
    else
        drop 1
    then
;
```

Advanced Features

Variables

Create variables that store values in memory:

```
VARIABLE counter
10 counter !     \ Store 10 in counter
counter @ .      \ Print value of counter
```

Constants

Create named constants:

```
42 CONSTANT answer
answer .         \ Prints 42
```

Memory Management

Word	Stack Effect	Description
`cells`	`( n -- bytes )`	Convert cells to bytes
`allot`	`( n -- )`	Allocate n cells of memory

Loop Indices

In do-loops, access the current index:

Word	Stack Effect	Description
`i`	`( -- index )`	Get current loop index
`j`	`( -- index )`	Get outer loop index

Examples

Basic Arithmetic

```
5 3 + .        \ Prints 8
10 3 - .       \ Prints 7
4 5 * .        \ Prints 20
15 4 / .       \ Prints 3
15 4 mod .     \ Prints 3
```

Stack Manipulation

```
5 dup . .      \ Prints 5 5
1 2 3 .s       \ Prints < 1 2 3 >
1 2 swap .s    \ Prints < 2 1 >
1 2 over .s    \ Prints < 1 2 1 >
```

User-Defined Functions

```
: add dup + ;     \ Double a number
5 add .           \ Prints 10

: greet ." Hello, World!" cr ;
greet             \ Prints Hello, World!
```

Control Flow Examples

```
\ Conditional execution
10 5 > if ." Greater" then

\ Begin-while-repeat loop
0 begin dup 5 < while dup . 1 + repeat drop

\ Do-loop (prints 0 1 2 3 4)
5 0 do i . loop
```

Variables and Constants

```
VARIABLE x
42 x !
x @ .

100 CONSTANT max-value
max-value .
```

Error Handling

The interpreter provides non-fatal error handling. Common errors include:

- **Stack underflow/overflow**: Trying to pop from empty stack or push to full stack
- **Division by zero**: Attempting to divide by zero
- **Unknown word**: Referencing undefined words
- **Memory access**: Invalid memory addresses
- **Compilation errors**: Syntax errors in word definitions

When an error occurs:

1. An error message is printed
2. The interpreter state is reset
3. Execution continues

Testing and Debugging

Interactive Testing

Use `.s` frequently to inspect the stack:

```
5 3 + .s    \ Check stack after operations
```

Batch Testing

Create test files and run them:

```
./forth < test.forth
```

Debugging Tips

1. Use `.s` to monitor stack state
2. Test operations incrementally
3. Check for stack underflow errors
4. Verify control flow nesting
5. Use simple test cases first

Example Test File

```
\ Test basic arithmetic
5 3 + . cr
10 2 - . cr

\ Test stack operations
1 2 3 .s cr
dup .s cr
drop .s cr

\ Test user-defined word
: test 2 * 3 + ;
5 test . cr
```

Note: Comments (lines starting with `\`) are shown for clarity but are not currently supported by the interpreter. All text is treated as executable code.

Technical Details

Stack Size

- Data stack: 1024 cells
- Return stack: 1024 cells
- Dictionary: 256 words maximum

Data Types

- All values are 64-bit integers (`long long`)
- Memory addresses are also 64-bit

Performance

- Dictionary lookup: O(n) linear search
- Stack operations: O(1)
- Memory access: O(1)

Limitations

- Fixed memory sizes
- No floating-point arithmetic

- Simple dictionary implementation
- Line-based input processing

Conclusion

This Forth interpreter provides a solid foundation for learning stack-based programming concepts. Start with simple arithmetic and stack operations, then progress to user-defined words and control flow constructs. Experiment freely and use the REPL to explore the language interactively.