# MP4 - Forth

## Objectives

The objective of this MP is to implement a stack-based language. Such languages are often useful for embedded systems. (Postscript, the printer language, is an example.) The language we will implement is a simplified version of stack based programming language called Forth.

## Goals

- Simulate a stack using recursive function calls
- Create a function lookup table
- Distinguish between an interpreter and a compiler
- See a new language paradigm (stack based) and have fun with it.

## Readings

To get a rough idea about Forth, you can start from following links:

- Starting Forth is the classic tutorial to Forth programmming language.
- Gforth is the GNU project that implements Forth interpreter in C. You can dowload and try it locally. **This MP will mainly follow Gforth for output and error messages.**
- Easy Forth provides a short tutorial and a simplified Forth interpreter (still more complicated than this MP) in JavaScript so that you can try it online.
- Forth Tutorials provides a list of references to know more about Forth.

Note that we are implementing a greatly reduced version of the Forth language. The behavior of our interpreter may not exactly match Easy Forth or Gforth.

# Getting Started

## How Forth Works

Forth is a stack-based language. All user-input is split on white-space into a sequence of *words*. The words are checked one at a time, and stack is modified accordingly; they are interpreted according to the dictionary. Built-in operators are supported by initializing the dictionary with predefined entries.

The Forth interpreter/compiler follows the work-flow below.

Interpreter mode:

- First, Forth checks if the word is in the dictionary of special symbols.

    - If the word is the ":" operator, Forth enters compiler mode immediately to add the new definition to the dictionary. After compiler mode is finished, it evaluates the rest of the words
    - Otherwise, it loads the definition and runs it.

- If the word is not in the dictionary, Forth checks if the word is an integer; in that case the integer is pushed to the integer stack.

- If the word is neither an integer nor a dictionary word (or one of a few special operators), the integer stack is emptied, the input stream is flushed, and an error message is printed.

```
> 2 5 furbitz
Undefined symbol: 'furbitz'
>  : furbitz 1 + ;
ok
> 1 furbitz
ok
> bye
Bye!
```

Compiler mode:

- Forth takes the word immediately following ":" as the identifier for the new definition.
- Next Forth steps through the remaining words, chaining them together to form a single instruction…
- Until either:
    - Forth finds a ";" word signalling the termination of compiler mode, or
    - Forth reaches the end of line and reports an error

Note: For compiler mode, the definition should be compiled to machine instructions and therefore varies from one hardware architecture to another. In this MP, we benefit from our meta-language Haskell, so we *compile* the definition to a continuation that encapsulates a sequence of stack operations and state transitions. We will describe the compiler in detail in Problem section.

## Relevant Files

In the directory `src` you'll find `Lib.hs` with all the relevant code. In this file you will find all of the data definitions, the primitive function maps, some stubbed-out simple parsers for special inputs, stubbed out evaluation functions, and the REPL itself.

## Running Code

To run your code, start GHCi with `stack ghci` (make sure to load the `Main` module if `stack ghci` doesn't automatically). From here, you can test individual functions, or you can run the REPL by calling `main`. Note that the initial `$` and `>` are prompts.

```
$ stack ghci
 ... Some Output ...
*Main> main
Welcome to your forth interpreter!
> 10 32 + .
42
ok
> 2 3 + 5 6 + + .
16
ok
> bye
Bye!
```

To run the REPL directly, build the executable with `stack build` and run it with `stack exec main`.

## Testing Your Code

You will be able to run the test-suite with `stack test`:

```
$ stack test
```

It will tell you which test-suites you pass, fail, and have exceptions on. To see an individual test-suite (so you can run the tests yourself by hand to see where the failure happens), look in the file `test/Spec.hs`.

You can run individual test-sets by running `stack test --ta "-t TEST-PATTERN"` where TEST-PATTERN specifies the pattern to search for test properties or test groups. All tests within matching groups or properties will be executed.

```
$ stack test --ta "-t dup"
mp4-forth-0.1.0.0: test (suite: test, args: -t dup)

=G= Dictionary for primitive operators:
  =P= IStack Manipulations `dup`: [OK, passed 100 tests]

         Properties  Total
 Passed  1           1
 Failed  0           0
 Total   1           1
```

```
$ stack test --ta "-t Operators"
mp4-forth-0.1.0.0: test (suite: test, args: -t Operators)

=G= Dictionary for primitive operators:
  =P= Arithmetic Operators: [OK, passed 100 tests]
  =P= Comparison Operators: [Failed]
*** Failed! (after 1 test):
Exception:
  Prelude.undefined
  CallStack (from HasCallStack):
    error, called at libraries/base/GHC/Err.hs:79:14 in base:GHC.Err
    undefined, called at src/Lib.hs:107:14 in
mp4-forth-0.1.0.0-f6OEElXAAq28J3aJyjkjH:Lib
Given operator: "<"
Given IStack (Top at left):
[1,-3]
(used seed 5369197114094880 64)

         Properties  Total
 Passed  1           1
 Failed  1           1
 Total   2           2
```

Look in the file `test/Spec.hs` to see which test properties were tested.

# Given Library Code

The setup code is concerned with importing the modules we will need and declaring the types we will use.

```haskell
-- for stack underflow errors
msgUnderflow :: String
msgUnderflow = "Stack underflow"

-- ... Other predefined error messages

-- for stack underflow errors
underflow :: a
underflow = error msgUnderflow
```

## The Types

The Forth machine will need to keep track of its state (`ForthState`). It will have an integer stack for intermediate results (`IStack`), a dictionary for primitive and defined functions (`Dictionary`), and a place to keep track of output messages (`Output`). State to state functions are defined as transitions (`Transition`).

```haskell
type ForthState = (IStack, Dictionary, Output)
type Transition = (ForthState -> ForthState)
type IStack     = [Integer]
type Dictionary = [(String, Value)]
type Output     = [String]

type CStack     = [(String, Transition)]
```

The dictionary will store `(key, value)` pairs where a key is a `String` that can be primitive operators, reserved keywords, or user defined function names. The mapped values (`Value`), which can be either primitive state transitions (`Prim :: Transition -> Value`) or commands only used in compiler mode (`Compile :: (CStack -> Maybe CStack) -> Value`).

We also have data-constructors for numbers (`Num :: Integer -> Value`) and unknowns (`Unknown :: String -> Value`). This is so that the dictionary lookup function `dlookup` can signal the `eval` and `compile` function that the lookup failed, and the input is not a number.

We also provide a `Show` instance for `Value`, which allows us to pretty-print things of type `Value`.

```haskell
data Value = Prim Transition
           | Define
           | EndDef
           | Compile (CStack -> Maybe CStack)
           | Num Integer
           | Unknown String

instance Show Value where
    show (Prim f)    = "Prim"
    show (Compile _) = msgCompileOnly
    show Define      = "Define"
    show EndDef      = msgCompileOnly
    show (Num i)     = show i
    show (Unknown s) = msgUndefinedSym s
```

## Dictionary Access

### Lookups

When `eval` uses `dlookup` and the lookup succeeds (the entry is present), the most recent definition in the dictionary will be used (closer to the head). If the lookup fails, then `dlookup` will try to convert the input to an `Integer` using `reads`. If that fails, it will signal `eval` that nothing worked by using the data-constructor `Unknown`.

```haskell
-- handle input lookups (and integers)
dlookup :: String -> Dictionary -> Value
```

```
dlookup word dict
    = case lookup word dict of
        Just x -> x
        _          -> case reads word of
                          [(i,"")] -> Num i
                          _        -> Unknown word
```

**Insert**

On inserting, we will simply add the new definition (Value) to the front of the dictionary. In this way, you preserve past definitions in case you ever extend our language with the ability to revert to previous definitions. Our language will not have that extension yet, but Forth language includes the concept of **marker** to store current state and discard definitions newer than a given marker. You can think about how to add it.

```
-- handle inserting things into the dictionary
dinsert :: String -> Value -> Dictionary -> Dictionary
dinsert key val dict = (key, val):dict
```

# Given Executable Code

Initial State, Read-Eval-Print Loop, and the `main` function are implemented in `app/Main.hs`. These code provides the interactive interpreter described in Running Code. You wouldn't need to modify or refer to them in order to finish this MP.

## Initial State

The initial state will have an empty integer stack (`initialIStack`), a dictionary with initial function definitions (`initialDictionary`), and an empty output stack (`initialOutput`).

The `initialDictionary` is defined later in the Problems section because you must complete it as part of the assignment.

```
import System.IO (hFlush, stdout)
import Lib (IStack, ForthState, initialDictionary, eval)

-- initial integer stack
initialIStack :: IStack
initialIStack = []

-- initial output
initialOutput :: [String]
initialOutput = []
```

```haskell
-- initial ForthState
initialForthState :: ForthState
initialForthState = (initialIStack, initialDictionary, initialOutput)
```

### Read-Eval-Print Loop

The read-eval-print loop handles all the action. It puts a prompt on the screen, reads some input, spits the input into words, feeds the words to the evaluator, and keeps track of the updated state. It also outputs anything that `eval` says should be output. Notice how the output is reversed before printing it because it is built in reverse order when processing the input recursively.

```haskell
repl :: ForthState -> IO ()
repl state
    = do putStr "> "
         input <- getLine
         hFlush stdout
         if input == "bye"
            then do putStrLn "Bye!"
                    return ()
            else let (is, d, output) = eval (words input) state
                 in  do mapM_ putStrLn (reverse output)
                        repl (is, d, [])

main = do putStrLn "Welcome to your Forth interpreter!"
          repl initialForthState
```

## Problems

### Lifters

Since a large portion of the built-in operators in Forth is modifying only the `IStack` instead of the whole `ForthState`. We provide the following lifter function to lift a stack operation to a Forth state transition.

Notice that we invoke `underflow` error when the lifted function returns `Nothing` because the stack operation failed.

```haskell
liftIStackOp :: (IStack -> Maybe IStack) -> ForthState -> ForthState
liftIStackOp op (i, d, o)
    = case op i of
        Just i' -> (i', d, o)
        Nothing -> underflow
```

#### `liftIntOp`

To make our lives easier, we will use a function `liftIntOp` that takes a Haskell function and converts it into one that will work on the integer stack. Essentially, the function pops out the top two elements in current `IStack`, calculates the result with given `op`, and push the result back to `IStack`. This will be used to create primitive operators, which will be stored in the `initialDictionary` (see next problem for example).

Note the order that the arguments are given to the operator `op` in. Also note that we return `Nothing` if there are not enough entries on the input stack.

```
liftIntOp :: (Integer -> Integer -> Integer) -> IStack -> Maybe IStack
liftIntOp op (x:y:xs) = Just $ (y `op` x) : xs
liftIntOp _  _        = Nothing
```

#### `liftCompOp`

You need to define `liftCompOp` so that we can have comparison operators between integers in our Forth language. In Forth, `0` is false and anything else is true, so you must return `0` if the result is `False`. If the result is `True`, you should return `-1` (following Forth tradition).

```
liftCompOp :: (Integer -> Integer -> Bool) -> IStack -> Maybe IStack
liftCompOp = undefined
```

### The Dictionary

As mentioned in previous sections, the interpreter supports built-in operators via a dictionary initialized with predefined entries. Here we are defining these entries for `initialDictionary`. To help you define other built-in operators, we provided the first entry for `+` in the `initialDictionary`. You must finish the rest of the definitions. Notice how we use `liftIntOp` to turn the Haskell function `(+)` into one that works on our Forth integer stack. The result type `IStack -> Maybe IStack` indicates that the function should return `Nothing` when underflow occurs. We further lift the function using `liftIStackOp`, so it could be applied on `ForthState` and throw `underflow` when receiving `Nothing`. Then we wrap the function in a `Prim :: (ForthState -> ForthState) -> Value` so that we can hold it as a value in a `Dictionary`.

```
initialDictionary :: Dictionary
initialDictionary = initArith ++ initComp ++ initIStackOp ++ initPrintOp
                 ++ initCompileOp
```

#### Arithmetic Operators

Provide the definition of the arithmetic operators subtraction (`"-"`), multiplication (`"*"`), and integer division (`"/"`). You will find the function `liftIntOp` useful here. We have provided addition (`"+"`) for you.

Note: You are encouraged to define division that generates a Forth error rather than a Haskell error, but it will not affect your grade.

### Comparison Operators

Provide the definition of the comparison operators less-than (`"<"`), greater-than (`">"`), less-than-or-equal-to (`"<="`), greater-than-or-equal-to (`">="`), equal-to (`"="`), and not-equal-to (`"!="`). You will find the function `liftCompOp` useful here.

Note: equal-to (`"="`) and not-equal-to (`"!="`) operators in Forth are different from Haskell.

### Stack Manipulations

Define `swap` (which swaps the top two elements), `drop` (which pops the top element without printing), and `rot` (which pops the third element and pushes it to the top of the IStack).

We have provided the definition of `dup` which duplicates the top of stack.

Make sure you handle the cases when stack underflow happens.

```
> 2 3 4 .S
2 3 4
ok
> dup .S
2 3 4 4
ok
> drop .S
2 3 4
ok
> swap .S
2 4 3
ok
> rot .S
4 3 2
ok
> 3 dup rot .S
4 3 3 3 2
ok
```

### Popping the Stack

For printing and supporting more advanced operators, we now have to modify more than just the `IStack` inside a `ForthState`.

Here, we handle one Forth word for you, the `.` operator. This operator consumes one element of the integer stack and outputs it. Notice once again how we handle the underflow case by generating the `underflow` error in `printPop` function. The mapping from `.` to `printPop`is then added into `initialDictionary`.

```
printPop :: ForthState -> ForthState
printPop (i:istack, cstack, dict, out) =
    (istack, cstack, dict, show i : out)
printPop _ = underflow
```

### Printing the Stack

Define `.S` which prints the entire stack. It does not consume the stack, however. It should print from bottom of stack to top. You may find the built-in Haskell function `unwords` useful here.

```
> 2 3 4 .S
2 3 4
ok
>
```

## Evaluator

Next is the evaluator. It takes a list of strings as the next tokens of input, a Forth state, and returns a Forth state. We have handled the implementation for you. Be sure you understand the code!

```
eval :: [String] -> ForthState -> ForthState
```

If the input is empty, we are done processing input and should return the current Forth state with `ok` indicating we successfully interpret the given words.

```
-- empty input -> return current state and output "ok"
eval [] (istack, dict, out) = (istack, dict, "ok":out)
```

### Lookup in dictionary

Otherwise, we check if the word matches one of the defined words in dictionary. `eval` will use `dlookup` to determine what to do with it. It could be that we get a number. If so, push it onto the stack. It could be that we get a built-in or user defined primitive function. If so, modify the state by feeding it to the function. It could also get `Define` indicating the beginning of a user definition, so we invoke `compileDef` to compile and update the dictionary.

If instead `dlookup` says that its an `Unknown` or `Complie` other than `Colon`, we then empty the `IStack`, flush the input stream, and output that error message accordingly.

```
-- otherwise it should be handled by `dlookup` to see if it's a `Num`, `Prim`,
-- `Define`, or `Unknown`
eval (w:ws) state@(istack, dict, out)
    = case dlookup w dict of
```

```
Prim f    -> eval ws (f state)
Num i     -> eval ws (i:istack, dict, out)
Define    ->
    case compileDef ws dict of
        Right (rest, dict') -> eval rest (istack, dict', out)
        Left msg -> ([], dict, msg:out)
otherwise -> -- reset IStack and add error message
    ([], dict, (show otherwise):out)
```

## Compiler

Consider the operators we defined for evaluation so far, all these operations directly modifies `IStack` or `Output`; therefore they can be interpreted right away. However, we would also like some common language features in other languages, such as user defined functions/procedures for code reuse and modularity, conditional decisions to alter program flow, and loops for repetition. These features are achieved more easily by *compiling* the computations instead of interpreting them right away, and *executing* the computations later when used. This concept is very similar to **continuation** in functional programming. In fact, we will ask you to "compile" a user definition in Forth to a continuation in Haskell for this MP.

In this MP, we follow Gforth to implement user definitions. A user definition starts from a colon `:` and ends with a semicolon `;`. Also a set of reserved words, namely `for`, `next`, `if`, `else`, `then`, `begin`, and `until`, is specified as compile-only words in `initialDictionary`. These reserved words are only allowed within user definitions; therefore they must be enclosed with in a pair of `:` and `;`. `eval` should output error messages when these words are used, and we will now implement functions to compile these reserved words as well as other words we've defined for Evaluator.

### User definitions

Now we are ready to add something interesting. Add the ability to define new words. The syntax is

```
: <name> <definition...> ;
```

The definition will be compiled as a continuation `k` with type `Transition`, in other words, `ForthState -> ForthState`. Then `(<name>, Prim k)` will be added into the dictionary of the Forth state. In the future when the symbol `<name>` is encountered, the definition will be looked up in the dictionary, so `eval` can find `k` and apply `k` on current `ForthState` just as other primitive operators. See below for how this is handled.

For example, to make the square function:

```
> : square dup * ;
ok
```

```
> 4 square .
16
ok
```

We get the name `square` and the definition body `dup * ;`. To compile the body, we first start from the initial continuation `id` that given a `ForthState` it returns the same state. Then we lookup the dictionary and find that `dup` maps to `Prim (liftIStackOp istackDup)`, so we can construct a new continuation `k1` by `\state -> (liftIStackOp istackDup) (id state)` or equivalently `(liftIStackOp istackDup) . id` to accumulate computations.

Similarly, we lookup `*` and find it maps to a value `Prim f2`. We then can construct the continuation `k2` with `\state -> f2 (k1 state)` or equivalently `f2 . k1` meaning that we apply `k1` then `f2` on any given state. Finally, we reach the end of definition `;`, so we update the dictionary with the entry `("square", Prim k2)`.

To handle well-nested control structure for compilation, we introduce `CStack` and `Compile` values. Notice in the code below, continuation of primitive operators (`Prim f`) are accumulated on the current top of `CStack` via `updateTop`. To deal with keywords for compile-only, we lookup from dictionary to get `cf` to update `cstack`. We will discuss how `cf` works by giving an example of handle `for ... next` loop in next section.

The implementation is given below. Be sure you understand how the code works!

```haskell
-- | Return rest of words after compilation and a dictionary w/ new defintion
-- Assuming ':' is already striped away
compileDef :: [String] -> Dictionary -> Either ErrorMsg ([String], Dictionary)
compileDef [] _ = Left msgZeroLenDef
compileDef (name:ws) dict
    = case  compile ws dict [("", id)] of
       Right (rest, f) -> Right (rest, dinsert name (Prim f) dict)
       Left msg -> Left msg

compile :: [String] -> Dictionary -> CStack
           -> Either ErrorMsg ([String], Transition)
compile [] _ _ = Left "The definition does not end"

compile (w:ws) dict cstack
    = case dlookup w dict of
        Prim f     -> compile ws dict (updateTop f cstack)
        Num i      -> compile ws dict (updateTop f cstack)
                      where f = (liftIStackOp (\is -> Just (i:is)))
        Compile cf-> case cf cstack of
                Just cstack' -> compile ws dict cstack'
                Nothing -> Left $ msgUnstructured w
        Define     -> Left "Nested definition is not allowed"
```

```
        EndDef    -> case cstack of
            [("", k)] -> Right (ws, k)
            otherwise -> Left $ msgUnstructured w
        otherwise -> Left $ show otherwise

updateTop :: Transition -> CStack -> CStack
updateTop k ((c, kold):cs) = (c, k . kold) : cs
updateTop _ [] = underflow
```

### Definite Loops

To help you understand how to use `CStack` to deal with well-nested language constructs, we give the implementation for one of the simplest loop structure in Forth as follows:

```
for <loop-body> next
```

The semantics of this language construct is, when `for` is encountered, it pops the top of `IStack` and get a number `i`; `<loop-body>` is then executed exactly `i+1` times (i.e., from `0` to `i`) if `i` is non-negative. Otherwise when `i` is negative, we simply don't execute `<loop-body>` in this MP.

```
> : incTwice 1 for 1 + next ;
ok
> 40 incTwice .
42
ok
> : incN+1 for 1 + next ;
ok
> 31 10 incN+1 .
42
ok
```

To compile `for ... next` to a `Transition`, we need the computation of `<loop-body>` alone so that we can repeat it. Therefore, in `cstackFor`, we push `("for", id)` into `CStack` to accumulate continuations only for `<loop-body>`. Recall `updateTop` function, any succeeding primitive function will now be composed in this new top element. Also any computation before `for` is still in `CStack`.

Up until encountering `next`, `cstackNext` is called by looking up dictionary. We know that the top element in `CStack` should match `("for", kloop)`, or else the loop is unstructured. In addition, the second element for top, `(c, kold)` should preserve computation right before `for`. Here, we design an auxiliary function `transForLoop` to help compose a new `Transition` from `kloop`. It is obvious that `transForLoop` checks if the top `i` of `IStack` in a given `ForthState` is negative and return the state with `i` popped. If `i` is non-negative then it compose `kloop` with itself `i` times via `aux` function and apply it on the state.

Finally, the return result from `transForLoop` is composed with `kold` to update the current top of `CStack`.

```
cstackFor :: CStack -> Maybe CStack
cstackFor cstack = Just $ ("for", id):cstack

cstackNext :: CStack -> Maybe CStack
cstackNext (("for", kloop):(c, kold):cstack) =
    Just ((c, knew):cstack) where knew = (transForLoop kloop) . kold
cstackNext _ = Nothing

transForLoop :: Transition -> (ForthState -> ForthState)
transForLoop kloop (i:is, d, o) =
    if i < 0 then
        (is, d, o)
    else
        (aux kloop i) (is, d, o)
        where aux k 0 = k
              aux k n = k . (aux k (n-1))
transForLoop _ _ = underflow
```

### Conditionals

Add conditionals. The syntax for conditions is

```
if <if-branch> else <else-branch> then
```

The `else <else-branch>` is optional.

The keyword order looks a bit different than in the languages you have been using. When you read an `if`, the top element is popped from `IStack` and compared with 0. If the element is not equal to 0, then `<if-branch>` is taken, or else `<else-branch>` is taken.

Notice how nested `if ... else ... then` statements should be handled properly in the examples below.

```
> : f 3 4 < if 10 else 20 then . ; f
10
ok
> : f 3 4 > if 10 else 20 then . ; f
20
ok
> : f 3 4 > if 10 then . ; f
main: Stack underflow
... More error messages.

> : f 3 4 < if 10 then . ; f
10
```

```
ok
> : f 3 4 < if 3 4 < if 10 else 20 then else 30 then . ; f
10
ok
> : f 3 4 < if 3 4 > if 10 else 20 then else 30 then . ; f
20
ok
> : f 3 4 > if 3 4 < if 10 else 20 then else 30 then . ; f
30
ok
```

In this problem, you have to implement how to modify `CStack` for `if`, `else`, and `then`. The actual flow for compiling `if ... else ... then` is first compiling the continuation `kif` of `<if-branch>` after you see `if`, and compiling `kelse` of `<else-barnch>` separately after you see `else`. Eventually, when you see `then`, the final continuation is constructed using `kif`, `kelse`, and the continuation `kold` that represents any continuation before this branch.

```
cstackIf :: CStack -> Maybe CStack
cstackIf cstack = undefined

cstackElse :: CStack -> Maybe CStack
cstackElse cstack@(("if", _):_) = undefined
cstackElse _ = Nothing

cstackThen :: CStack -> Maybe CStack
cstackThen (("else", kelse):("if", kif):(c, kold):cstack) = undefined
cstackThen (("if", kif):(c, kold):cstack) = undefined
cstackThen _ = Nothing
```

### Indefinite Loops

Another loop structure in Forth is as follows:

```
begin <loop-body> until
```

The part between `begin` and `until` is executed repeatedly until the top element consumed by `until` is True (not equal to 0). For example, `noLoop` will always stop at the first iteration because `until` will consume `-1`. `infLoop` will loop forever because `until` will consume `0` every time. `toZero` will repeat until the top element is less than or equal to 0.

```
> : noLoop begin -1 until ;
ok
> noLoop
ok
> : infLoop begin 0 until ;
ok
```

```
> infLoop
... It won't stop. Press Ctrl+C to interrupt.

> : toZero begin .S 1 - dup 0 <= until ;
ok
> 4 toZero
4
3
2
1
ok
```

Again you use `CStack` to handle loops. When you hit a `begin`, you should start accumulate continuation for `<loop-body>` on top of `CStack`. Once you hit `until`, you will have the continuation `kloop` for `<loop-body>` and `kold` that represent the continuation before the loop. Use `kloop` and `kold` to construct the final continuation.

```haskell
cstackBegin :: CStack -> Maybe CStack
cstackBegin cstack = undefined

cstackUntil :: CStack -> Maybe CStack
cstackUntil (("begin", kloop):(c, kold):cstack) = undefined
cstackUntil _ = Nothing
```