Forth Golfscript Interpreter

Golfscript

Golforth

Code Golf

- shortest possible source code that implements an algorithm
- solving problems (holes) in as few keystrokes as possible

Golforth

Code Golf

- shortest possible source code that implements an algorithm
- solving problems (holes) in as few keystrokes as possible

Golfscript

- stack oriented, variables exist
- single symbols represent high level operations
- strong typed
- heavy use of operator overloading and type coercion

Golforth

Golfscript Types

- Integer: `1 2`
- Arrays: `[1 2 3] [3]`
- Strings: `"one two three"`
- Blocks: `{1+}`

Golforth

Golfscript Types

- Integer: `1 2`
- Arrays: `[1 2 3] [3]`
- Strings: `"one two three"`
- Blocks: `{1+}`

Golfscript Operator Example

- `12 3 * -> 36`
- `[50 51 52]' '* -> "50 51 52"`
- `[1 2 3]{1+}/ -> 2 3 4`
- `{.@\%.}do; ( n1 n2 -- gcd )`

Forth Implementation

Golforth

Typesystem

- Values as scalar references on stack
- Anonymous functions vs Memory

  -
    ```
    : anon_int { u -- typext }
    :noname u POSTPONE LITERAL POSTPONE typeno_int POSTPONE ; ;
    ```

  -
    ```
    : anon_int { u -- addr }
    2 cells allocate IF
      abort
    ELSE
      tuck ! typeno_int over cell+ !
    ENDIF
    ```

```
12 anon_int s" 1 anon_int golf_+" anon_block
```

$$\downarrow$$

`2 elements on stack (12 and {1+})`

Golforth

Arrays

- Construction similar to postscript.
- **[** marks stack size, **]** collects back to marked size.
- Mark moves when stack becomes smaller:

```
1 2 [\] -> [2 1]
```

Golforth

Arrays Implementation

```
: golf_slice_start ( -- )
  depth slice_start ! ;

: anon_array ( x1 ... xn -- array )
  depth slice_start @ - dup >r
  dup cells dup allocate
  + swap 0 u+do
    cell tuck !
  loop r>
  ...;

[1 3 5] -> golf_slice_start 1 anon_int 3 anon_int 5 anon_int anon_array
```

Golforth

Blocks

- Stored as already translated strings
- Operations: `2{1+}+` $\rightarrow$ `{2 1+}`
- Execution via `evaluate`

Golforth

Parser

- translates golfscript to forth based intermediate strings
- based on regular expression of reference implementation
- Responsible for:
  - infer initial type from syntax
  - symbol table for variable tracking
  - note that every value can be a variable!

```
"2 {1+}:x"
```

$$\downarrow$$

(creating x in symbol table)

$$\downarrow$$

```
2 anon_int s" 1 anon_int golf_+" anon_block x ,
```

Golforth

Type Coercion and Overloading

- Typeorder for Coercion
- Coercion according to highest order type
- Heavy operator overloading results in wide range of functionality

Golforth

```
*: Multiplication
    2 4* -> 8

*: Execute a block a certain number of times
    2 {2*} 5* -> 64

*: Array/string repeat
    [1 2 3]2* -> [1 2 3 1 2 3]
    3'asdf'* -> "asdfasdfasdf"

*: Join
    [1 2 3]','* -> "1,2,3"
    [1 2 3][4]* -> [1 4 2 4 3]

*: Fold
    [1 2 3 4]{+}* -> 10
    'asdf'{+}* -> 414
```

Golforth

Conditionals and Loops

- `5{1-..}do` → 4 3 2 1 0 0
- `5{.}{1-.}while` → 4 3 2 1 0 0
- `5{.}{1-.}until` → 5
- implemented as words which consume code blocks

```
: golf_do { block }
    BEGIN
        block golf_execute
    WHILE
    REPEAT ;
```

Golforth

Cutbacks

- Error Handling differs
- Probably not all operators implemented

Golforth

Usage of Idiomatic Forth

- Stack paradigma mapped to typed language
- Wordlists for variable tracking
- Macros & anonym functions for language implementation
- Macros for operator implementation