Contents

Deoxyribose

The DNA-themed programming language all the kids are talking about

Introducing Deoxyribose

Deoxyribose is a stack-based esoteric programming language based on the syntax and function of DNA.

A valid deoxyribose program is made up entirely of the letters A, C, G, and T, which form three-digit **codons** in accordance with the DNA codon table. These codons correspond to stack or flow-control operations. All other characters are ignored, making the language suitable for polyglots and other such fun things.

Interestingly, the correspondence between codons and operations is **degenerate**, so there's more than one way to write each operator, just like in the real genetic code.

Execution of Deoxyribose takes place on a closed circle of DNA (like the bacterial genome). Any amount of information can be placed before the first start codon, and if no stop codon is found, execution will loop around to the very beginning. Clever use of the frameshifts this can lead to, combined with the degeneracy of the codon–amino acid correspondence, can allow some very fun and exciting things.

The name of the language was selected due to being the "most-DNA-y" word that doesn't contain an A, C, G, or T, meaning it can be included in a comment.

Amino acids (operators)

The operators are defined by their correspondence to particular amino acids, as follows.

Special codons

- `ATG`: Start program execution. Everything before this codon is ignored, and it is expected that the next codon will define the "block size" (see below); this codon also codes for methionine
- `TAG`, `TAA`, and `TGA`: Stop; exit the program and return 0.

Charged amino acids — Single-stack operations

- **His**: *Push* next block (expressed in quaternary notation, see below) to the top of the main stack
- **Lys**: *Pop* top of main stack to standard output as a number
- **Arg**: *Pop* top of main stack to standard output as a Unicode character
- **Glu**: *Dupe* (duplicate) the top element of the main stack
- **Asp**: *Drop* the top element of the main stack

Non-polar amino acids — Two-stack operations

- **Leu**: *Plus* (add) the top elements of the main and auxiliary stacks, remove these elements, and place the result on top of the main stack
- **Ile**: *Minus* (subtract) the top element of the auxiliary stack from the top element of the main stack, remove these elements, and place the result on top of the main stack
- **Val**: *Mul* (multiply) the top elements of the main and auxiliary stacks, remove these elements, and place the result on top of the main stack
- **Pro**: *Divide* the top element of the main stack by the top element of the auxiliary stack, round the result towards zero, remove these elements, and place the result on top of the main stack
- **Met**: *Swop* (swap) the top elements of the main and auxiliary stacks
- **Phe**: *Join* (concatenate) the main and auxiliary stacks, leaving the auxiliary stack empty
- **Gly**: *Move* the top element of the main stack to the top of the auxiliary stack
- **Trp**: *Power* (exponentiate) the top element of the main stack to the power of the top element of the auxiliary stack, remove these elements, and place the result on top of the main stack
- **Ala**: *Modulo* — calculate the remainder when the top element of the main stack is divided by the top element of the auxiliary stack, remove both of these elements, and place the result on top of the main stack

Polar amino acids — Flow control

- **Cys**: *Jump* – Jump to the next occurrence of the next codon
- **Asn**: *Loop* – Jump backwards to the previous occurrence of the next codon
- **Ser**: *Jump if <=0* – If the top element of the main stack is less than or equal to zero, jump to the *next* occurrence of the next codon
- **Thr**: *Loop if <=0* – If the top element of the main stack is less than or equal to zero, jump to the *previous* occurrence of the next codon
- **Tyr**: *Jump if null* – If the main stack is empty, jump to the next occurrence of the next codon
- **Gln**: *Loop if null* – If the main stack is empty, jump to the previous occurrence of the next codon

Note that the length of these jumps need not always be a multiple of three. This can cause frameshifts, which are half the fun.

Integer literals

"But," I hear you cry, "How am I supposed to represent numbers using only the letters A, C, G, and T?"

"Simple," I reply. "You just express natural numbers in base-4 such that A = 0, C = 1, G = 2, and T = 3."

Since a codon can only be three characters long, this limits integer literals to the range `AAA = 0` to `TTT = 63`.

Once stored inside the stack, values are not subject to these same limits, and are treated just like ordinary Python numbers. Larger values, floats, and negative numbers can be constructed using mathematical operations.

Unicode characters can be stored as their character reference and converted back by arginine. There is no built-in string (or array) datatype, only ints ordered on the stack.

Examples

Note that the examples below include spaces to separate segments of the code for readability; these are unnecessary, ignored by the interpreter, and excluded from the given byte counts.

The multi-line explanations given below each example will not run as expected unless all A, C, G, and T characters (case-insensitive) are removed from the comments; all other comment characters are fine.

Hello, world!

`ATG CATGAC CACTTTCACGCCGGTTTA ... CATTTTCATAGCGGTTTG AGATATATTAATTTG A` (Truncated because it's long and boring, actually prints `Hd!`)

Accepts no input; prints `Hello, world!` to STDOUT.

```
ATG Start

CAT His     Push
GAC 33      (ASCII !)

CAC His     Push
TTT 63
CAC His     Push
GCC 37
GGT Gly     Move
TTA Leu     Plus (= 100, ASCII d)

...

CAT His     Push
TTT 63
CAT His     Push
AGC 9
GGT Gly     Move
TTG Leu     Plus (= 72, ASCII H)

AGA Arg     Pop as char
TAT Tyr     Jump if null
ATT
AAT Asn     Loop                    ATT
TTG                                 TGA      End
A
```

Outdated (version 2) examples

Infinite Fibonacci sequence `ATGAAC CATAACGAA GGT TGTGAATTAGGTATGGAAAAAA`

(40 B)

Accepts no input; prints an infinite series of newline-separated integers to STDOUT. The printed sequence starts at 2, but it would be trivial to add the expected `1\n1\n` at the expense of a few more bytes.

```
ATG Start                       .AT Asn     Jump back to Cys
AAC Block size = 1

CAT His     Push
AAC 1
GAA Glu     Dupe

GGT Gly     Move

TGT Cys     Destination of Asn
GAA Glu     Dupe
TTA Leu     Plus
GGT Gly     Move
ATG Met     Swop
GAA Glu     Dupe
AAA Lys     Pop as int
A..         ...loop to start
```

Cat

`ATGAAC TGTGGTTATAATCAA TTT TGTAGATATAATCA` (38 B)

Accepts any number of characters or Unicode codepoints as arguments; prints its input to the screen. If the input contains spaces, these will be stripped unless the string is wrapped in quotes. Input containing numbers is fine, but entirely numeric input will be converted into the corresponding Unicode character (e.g. input of `100` prints `d`).

```
ATG Start                                       ..A Gln     Destination of Tyr
AAC Block size = 1                              TGA Stop

TGT Cys     Destination of Asn
GGT Gly     Move
TAT Tyr     If main stack is empty, jump to Gln
AAT Asn     Jump back to Cys
CAA Gln     Destination of Tyr

TTT Phe     Join

TGT Cys     Destination of Asn
AGA Arg     Pop as char
TAT Tyr     If main stack is empty, jump to Gln
AAT Asn     Jump back to Cys
CA.         ...loop to start
```

Print integers from 1 to N

`ATGAAC GGTCATAAC TGTGAAAAACATAACGGTTTATTTGAAGGTGGT GAAATTAGT TAG ACTGATA` (67 B)

Accepts an integer N as input; prints all integers from 1 to N to STDOUT, separated by newlines.

```
ATG Start                               .AT Asn     Jump back to Cys
AAC Block size = 1

GGT Gly     Move
CAT His     Push
```

```
AAC 1

TGT Cys     Destination of Asn
GAA Glu     Dupe
AAA Lys     Pop as int
CAT His     Push
AAC 1
GGT Gly     Move
TTA Leu     Plus
TTT Phe     Join
GAA Glu     Dupe
GGT Gly     Move
GGT Gly     Move

GAA Glu     Dupe
ATT Ile     Minus
AGT Ser     If <= 0, jump to next Thr

TAG Stop

ACT Thr     Destination of Ser
GAT Asp     Drop
A..         ...loop to start
```

Truth machine

`ATGTGAGAAAAATCTAACTTA` (21 B)

Accepts one integer as input. If this value is less than or equal to 0, it is printed once before the program terminates. If the value is greater than 0, it is printed infinitely.

```
ATG Start                           TAA Stop
TGA Block size = 56                 TGT Cys     Destination of Asn
GAA Glu     Dupe               GAG Glu     Dupe
AAA Lys     Pop as int              AAA Lys     Pop as int
TCT Ser     If <= 0, jump to Thr    AAT Asn     Jump back to Cys
AAC Asn     Jump back to Cys
TTA                                 ACT Thr     Destination of Ser
```

This code is heavily golfed, making liberal use of frameshifts for maximum compression.

Execution starts at the initial ATG. There are no integer literals anywhere, so the block size is meaningless and can be used for other purposes — more on that later. The top stack element is printed, then the magic happens. * If the top stack element (the user input) is zero, we jump forward to the `ACT` formed by Asn and the following `TTA` (which would code for Leu, but is never actually read in this reference frame), resulting in a frameshift. Looping back to the start forms the stop codon `TAA`. * If the top stack element is 1, we continue reading to Asn, which searches backwards for a Cys. This is found in the `TGT` formed by the start codon and the block size, so we jump back there, frameshifted. We now enter a tighter, less complex loop, which simply prints the top stack element indefinitely.

As this demonstrates, frameshifts and degeneracy are powerful tools that allow the same base sequence to do two (or more!) totally different things.

Note that the code `ATGAAC TCT TGTCATAACAAAAAT ACTTAG` (30 B)

```
ATG Start
AAC Block size = 1
```

```
TCT Ser     If <= 0, jump to Thr

TGT Cys     Destination of Asn
CAT His     Push
AAC 1
AAA Lys     Pop as int
AAT Asn     Jump back to Cys

ACT Thr     Destination of Ser
TAG Stop
```

doesn't look like it would print on an input of zero, but actually does because of a frameshift: The `ACA` formed by the numeric literal 1 and the following Lys is recognised as the destination of the Ser → Thr jump, then the frame-shifted `AAA` immediately following is interpreted as a Lysine; the code then runs through a series of invisible operations (Ile, Leu, Arg) before looping back to the start and terminating on the `TGA` formed by the start and block size codons.

In fact, a numeric literal 1 cannot appear anywhere between Ser and Thr, since all `ACN` codons translate to Thr, causing a frameshift. If the number 1 is necessary, and a frameshift is undesirable, the value must be pushed to the stack before the conditional or constructed in some other way (e.g. by subtracting 2 from 3). This makes life more fun.

Primality test

`ATGAAC GAACATAAG TGT GAAGGTGGC GCT CATAACGGT AGT GAT GAATTTGGTTTA AAT ACT GATTTTGATGGTATT AGT CATAAAAAATAG ACT CATAACAA` (104 B)

Accepts one integer as input; prints 1 if prime, 0 if composite.

```
ATG Start                       ..A Lys     Pop as int
AAC Block size = 1              TGA Stop

GAA Glu     Dupe
CAT His     Push
AAG 2

TGT Cys     Destination of Asn

GAA Glu     Dupe
GGT Gly     Move
GGC Gly     Move

GCT Ala     Modulo

CAT His     Push
AAC 1
GGT Gly     Move

AGT Ser     If <= 0, jump to Thr

GAT Asp     Drop

GAA Glu     Dupe
TTT Phe     Join
```

```
GGT Gly     Move
TTA Leu     Plus

AAT Asn     Jump back to Cys

ACT Thr     Destination of Ser

GAT Asp     Drop
TTT Phe     Join
GAT Asp     Drop
GGT Gly     Move
ATT Ile     Minus

AGT Ser     If <= 0, jump to Thr

CAT His     Push
AAA 0
AAA Lys     Pop
TAG Stop

ACT Thr     Destination of Ser

CAT His     Push
AAC 1
AA.         ...loop to start
```

Notes etc.

This is very much a work in progress, and there are some very poor aspects of the design that will likely be changed in backwards-incompatible ways in future versions. I would welcome suggestions.

The specification and documentation for this project are dual-licensed under a CC-BY licence and the MIT licence (see the `LICENSE` file), and the interpreter etc. are under an MIT licence. I'd love to see them developed and improved upon, but it would be nice to see my name on there somewhere.