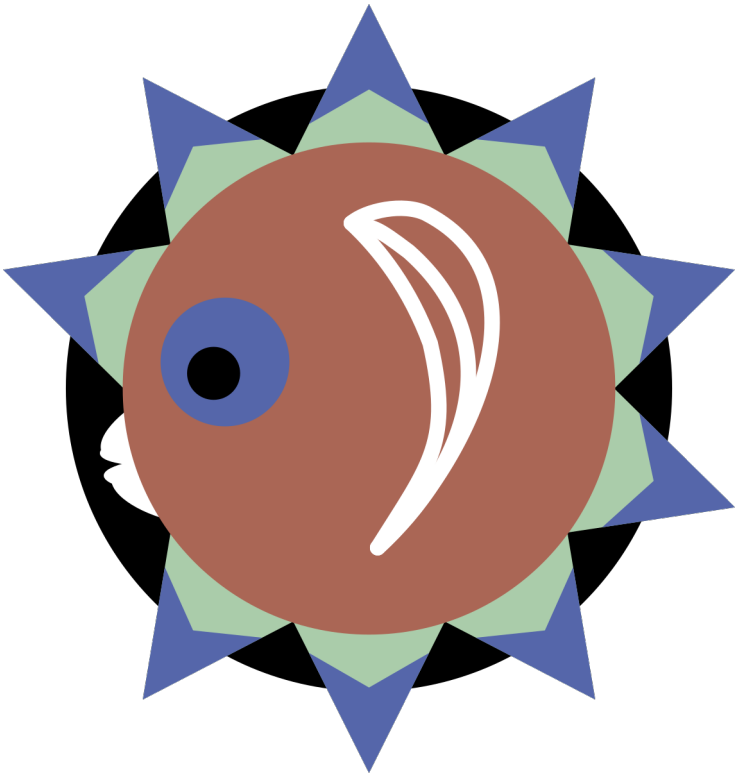
5b1t Documentation

Louis A. Burke



Contents

Introduction

This document provides documentation and instruction for how to write programs in 5b1t.

5b1t is a programming language designed to produce extremely terse code.

5b1t is written using the 95 printable ASCII characters, plus the new line character. The name derives from the fact that these characters can be encoded across 5 bits and 1 trit ($96 = 2^5 3$). Since when terseness is measured it is often done in bytes, 5b1t has a standard encoding across bytes that retains efficiency.

5b1t code cares deeply about types. Every type in 5b1t overloads nearly every operator. Due to the nature of 5b1t and its dependence on types it is compiled. Due to the number of built-in features it is compiled to another high level language which must in turn be compiled down to an executable. This differs from many other terse languages in that 5b1t is not an interpreted language.

Also due to the typing system of 5b1t, the initial compilation stage must compile for a specific input type. For instance a program compiled to take an integer as input might not behave the same as a program compiled to take a pair of integers as input. The type of the argument can be listed with the source separately, or the source can be assumed to take as argument an empty type, in which case it must consume and parse input itself.

In summary, the process of writing a 5b1t program involves encoding it, while running it involves decoding, compiling with type specifiers, recompiling, and then finally executing the resulting executable. While this process is more involved than most terse languages' process, it can be automated easily and produce executables that can run orders of magnitude faster than most interpreted languages.

Overview

5b1t is a concatenative language, like *Joy*. At all points the program maintains the 'current data' which is passed to various functions. Each symbol represents an operation. Depending on whether it is followed by a block of code, each operation may take a function as an additional argument. All operations work on the common state of execution.

Additionally, many 5b1t types inherit operations from each other, thus there are certain operations that are common to many or even all input types.

Control flow is represented using whitespace. A space character introduces a block while a newline terminates a block. A newline outside of a block terminates parsing and is treated as the end of the input. It may be helpful to think of the control structures similarly to quotations in the Joy programming language. A space/newline pair can be thought of as a [,] pair. If the program's whitespace is unbalanced, an appropriate number of balancing characters will be added. Thus if there are more spaces than newlines, that number of newlines will be added to the end. If there are more newlines than spaces, that number of spaces will be added to the start.

Example

An example program may help clarify control flow.

```
[1234m D
+
```

When run on the nothing type this is interpreted as follows.

First the `[` indicates a list constructor. Each subsequent number in a list constructor merely adds that number to the list. The result is the list of the numbers 1, 2, 3, and 4.

The `m` followed by a space indicates the map function. This will apply the rest of the line to the input list. The rest of the line contains `D` which, when applied to natural numbers doubles them. Thus, after applying the map, the current state is a list containing 2, 4, 6, and 8.

Finally `+` when applied to a list of natural numbers indicates the sum operation. Therefore the result of this code is 20.

Code Page

Below is the code page for 5b1t programs.

Bits	00000000000000001111111111111111
	00000000111111110000000011111111
	00001111000011110000111100001111
	00110011001100110011001100110011
Trits	01010101010101010101010101010101

0	0123456789ABCDEFGHIJKLMNOPQRSTUV
1	WXYZabcdefghijklmnopqrstuvwxyz+/
2	-*|^&%<>()[]{}.,;:'"`~!@#$?_=\␣¶

Encoding

While normally written in any superset of the ASCII character encoding, the true size of a 5b1t program is measured based on its encoded size. The encoding method is to first convert the input to a base-96 number, appending a 1 in front of it. This number is converted to binary and padded with leading zeroes to a multiple of 8 bits. The resulting bytes is the encoding.

Encoding Example. For example, consider the code `Hello, World!`. The code page expansion in base-96 is: `17 40 47 47 50 79 94 32 50 53 47 39 86`

After adding the leading 1 this yields: `1 17 40 47 47 50 79 94 32 50 53 47 39 86` which, in decimal, is `69494655939676481327868662`.

Converting this number to binary gives: `1110010111110000010000110110001110100001101100010100010011111111111001000 10101011110110`.

Padding this out to octets yields: `00111001 01111100 00010000 11011000 11101000 01101100 01010001 00111111 11110010 00101010 11110110`.

Therefore `Hello, World!` is encoded as `397C10D8E86C513FF22AF6` in hexadecimal.

Decoding Example. Another example, consider the hexdump of a program: `126F1B7809ED6F0BA8F7FB10AF90C516518F7D69665CB79E7158920855038D8A35CCA3F8EFE27EA6E668B3FF2EB08B152B27C97D4877E3EA1`.

Converting the base-16 number to decimal yields `837409650887258988666618071465997993605426886483341540932602321232594279790569864446464775538517817356961611860361920765020069565447841`. Which in base-96 is `1 33 5 0 86 25 69 87 10 25 74 4 93 25 35 33 5 4 72 25 67 73 7 12 12 73 7 77 89 14 18 12 10 27 64 28 29 10 23 13 10 27 13 64 10 23 29 18 31 18 27 30 28 64 29 14 28 29 64 15 18 21 14 86 89 17 62 17 65`.

Stripping the 1 and reversing the code-page yields:

`X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*`

Which is the source of the program.

Ramifications. The result of this encoding scheme is that the minimum number of bytes required to store a program are used. However, since each symbol takes over 6 bits ($\log_2 96 \approx 6.585$) most characters of input seem to take up a whole byte of output. The remaining bit and a half (approximately) of information accumulates until eventually an input character appears to be 'free' in that the total byte count does not increase. It may be useful to see a table of input length to output bytecounts highlighting the 'free' lengths.

Input	Output	Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
1	1	33	28	65	54	97	80	129	107	161	133
2	2	34	28	66	55	98	81	130	108	162	134
3	3	35	29	67	56	99	82	131	108	163	135
4	4	36	30	68	56	100	83	132	109	164	135
5	5	37	31	69	57	101	84	133	110	165	136
6	5	38	32	70	58	102	84	134	111	166	137
7	6	39	33	71	59	103	85	135	112	167	138
8	7	40	33	72	60	104	86	136	112	168	139
9	8	41	34	73	61	105	87	137	113	169	140
10	9	42	35	74	61	106	88	138	114	170	140
11	10	43	36	75	62	107	89	139	115	171	141
12	10	44	37	76	63	108	89	140	116	172	142
13	11	45	38	77	64	109	90	141	117	173	143
14	12	46	38	78	65	110	91	142	117	174	144
15	13	47	39	79	66	111	92	143	118	175	145
16	14	48	40	80	66	112	93	144	119	176	145
17	14	49	41	81	67	113	94	145	120	177	146
18	15	50	42	82	68	114	94	146	121	178	147
19	16	51	42	83	69	115	95	147	121	179	148
20	17	52	43	84	70	116	96	148	122	180	149
21	18	53	44	85	70	117	97	149	123	181	149
22	19	54	45	86	71	118	98	150	124	182	150
23	19	55	46	87	72	119	98	151	125	183	151
24	20	56	47	88	73	120	99	152	126	184	152
25	21	57	47	89	74	121	100	153	126	185	153
26	22	58	48	90	75	122	101	154	127	186	154
27	23	59	49	91	75	123	102	155	128	187	154
28	24	60	50	92	76	124	103	156	129	188	155
29	24	51	51	93	77	125	103	157	130	189	156
30	25	62	52	94	78	126	104	158	131	190	157
31	26	63	52	95	79	127	105	159	131	191	158
32	27	64	53	96	80	128	106	160	132	192	159

Language

This section defines every operation of the language. Each operation operates on the state of execution. Depending on the type of the state of execution, an operator may perform differing actions. Some types inherit their operations from other types.

What follows is a summary of the operations of each type. When a type is generic (that is, it can take types as parameters) the generic parameters are marked with enclosed numerics (e.g. (1) (2)). When a returned type is generic, enclosed alphabetics are used instead (e.g. (A) (B)). When an operator takes a function as a block it is represented by Λ.

The summaries are titled based on the type's name along with whatever type it inherits from, followed by its canonical representation.

Following the titles is a description of what a type represents and what it is used for. Often variable names are assigned in this section for future reference.

Finally a table summarizing the actions of each operator on the type is presented. The result type is shown in canonical representation and the description may make reference to the variables defined in the opening paragraph. Any as-yet unimplemented operations are coloured yellow while obsoleted or removed operations are coloured red and partially implemented operations are coloured blue. Inherited operations appear with grey text and merely direct you to their source.

Occasionally, after the table there will be additional notes. These are often to point out unusual or particularly interesting behaviour of certain operations.

Literals

Sometimes a command would like to take a literal value of some kind, however there are no literal values in 5b1t. To work around this there are a class of partially completed types marked with to indicate that they take a literal value.

The format of a literal value is an optional digit count followed by a base-94 number containing that many digits.

The digit count is made up of all symbols up to the first symbol in the first 64 codepoints. These have 64 subtracted from their values and are treated as a base-30 number. This number has 2 added to it to become the full digit count of the literal. If the digit count is omitted, then the number is assumed to have 1 digit. Note that this means the numbers from 64 to 95 take 3 symbols (e.g. 88 is `-0#`).

Any symbol with a block will be repeated as many times as the base-94 value of the block. This also applies to the base-94 symbols within said blocks.

Literal values print their initial argument if the result of the program is a partial application thereof.

Examples. A few examples may be useful to understand literals.

```
x
```

The literal value is 59 because it has a code-point of 59.

```
{Hello, World!
How are you?
Good.
Good.
```

This begins with `{` so it needs 14 base-94 digits. The `Hello` corresponds to $17 : 40 : 47 : 47 : 50$ in base-94, or 1360916352. The `,` takes a block containing `World!`. `World!` corresponds to $32 : 50 : 53 : 47 : 39 : 86$ in base-94, or 238797471964. The code-point of `,` is 79 therefore the partial value at this point is $17 : 40 : 47 : 47 : 50 : \underbrace{79 : \cdots : 79}_{238787471968}$ in base-94, with 238787471968 79s.

Returning to the consumption of the 14 base-94 digits, 8 are still missing. `Ho` corresponds to $17 : 50$ in base-94, or 1648. The `w` takes a block whose value must be calculated.

This block starts with `ar` corresponding to $36 : 53$ in base-94, but then the `e` has another block to calculate. This block contains `you?` with a value of $60 : 50 : 56 : 90$ in base-94, or 50282194. Thus the block of the `w` has a partial value of $36 : 53 : \underbrace{40 : \cdots : 40}_{50282198}$ in base-94. Now the remaining `Good.` is appended in base-94 to give the final repetition count on the `w` as $36 : 53 : \underbrace{40 : \cdots : 40}_{50282198} : 16 : 50 : 50 : 39 : 78$ in base-94.

Applying this to the `w` yields a partial result of:

$$17 : 40 : 47 : 47 : 50 : \underbrace{79 : \cdots : 79}_{238787471968} : 17 : 50 : \underbrace{58 : \cdots : 58}_{36:53:\underbrace{40 : \cdots : 40}_{50282198}:16:50:50:39:78}$$

Finally, the remaining `Good.` is appended showing that the base-94 value of the above literal is:

$$17 : 40 : 47 : 47 : 50 : \underbrace{79 : \cdots : 79}_{238787471968} : 17 : 50 : \underbrace{58 : \cdots : 58}_{36:53:\underbrace{40 : \cdots : 40}_{50282198}:16:50:50:39:78} : 16 : 50 : 50 : 39 : 78$$

This is a very contrived example, but shows just how massive literal values can be made with minimal code. In fact this number is far beyond the transcomputational limit, and thus would never actually be created in memory. It would take almost 100 million digits just to write out how many digits the number has. For now 5b1t uses GMP for bignum manipulation, which will fail to even store the result. In future it is hoped to use a more symbolic representation in order to enable the use of such large constants.

One can use the tool `to_literal` to create literal values easily.

Containers

There exists a convention for certain commonly used containers. There are four containers used by 5b1t. They are parameterized based on whether they care about order and how they treat types.

These containers are:

- Lists. Represented with square brackets.
- Sets. Represented with curly brackets.
- Tuples. Represented with round brackets.
- Groups. Represented with multiple datatypes in sequence with no bracketing.

Lists, sets, and tuples can all be created from the base datatype using the wrapping `)`, `]`, and `}` operators. Groups are far less common, arising from specific operations. Types created from groups are each effectively their own atomic type, as there is no means by which to index a group.

Order. Ordered containers have a well defined order. The elements can be indexed and re-ordered. Unordered containers are the opposite.

Type. Homogeneous containers are parameterized by a single type. This type is determined by finding the closest parent type that is a parent of all types in the list. In the base case these containers will contain objects so different that the only relevant parent type is the base datatype.

By contrast heterogeneous containers are parameterized by precisely which types are present. The container's type is determined by finding the closest type with the same number of elements and whose elements are each parents of their respective source elements.

Summary. With these definitions in mind, here are the containers in 5b1t.

	Ordered	Unordered
Homogeneous	List	Set
Heterogeneous	Tuple	Group

Examples. To determine the 'true' type can sometimes be difficult. As such, lets look at a few examples.

Lets pretend we only have four types other than the base datatype. Lets call these types A, B, C, and D. A and B will be direct descendants of the base datatype. Meanwhile C and D will be descendants of A.

Meanwhile lets pretend that the defined list types are A, B, and C lists (plus the base datatype list). Similarly lets pretend that the defined set types are A, B, and C sets (plus the base datatype set). Lets also pretend we have AA, AB, and AC tuples (plus the base datatype tuple). Similarly lets pretend that we have AA, AB, and AC groups (plus the base datatype group).

Now lets look at some examples. First lets consider a container with an A and a C. For each type this will yield the following.

	Ordered	Unordered
Homogeneous	A-List	A-Set
Heterogeneous	AC-Tuple	AC-Group

This is pretty straightforward.

Now lets consider a container with an A and a D. For each type this will yield the following.

	Ordered	Unordered
Homogeneous	A-List	A-Set
Heterogeneous	AA-Tuple	AA-Group

Finally lets consider a container with an A and a B. For each type this will yield the following.

	Ordered	Unordered
Homogeneous	List	Set
Heterogeneous	AB-Tuple	AB-Group

Types

Datatype() •_•. The datatype type is the root type of all other types. Any operation implemented by the datatype is inherited by every other type. Any operation not defined in the type hierarchy all the way back to the datatype will result in an unimplemented exception propagating to output.

Table 1: Operators for Datatype •_•

OP ↦ RES	Description
) ↦ (⊛)	Tuple Wrap: Returns a tuple containing only a single item, the input.
, ↦ (⊛Ⓐ)	Append: Returns a tuple of the input and the result of applying Λ to the empty datum.
. ↦ Ⓐ	Apply: Applies Λ to the state, returning the result.
; ↦ (Ⓐ⊛)	Prepend: Returns a tuple of the result of applying Λ to the empty datum and the input.
W ↦ (⊛⊛)	Duplicate: Creates a tuple containing the state twice.
Y ↦ Ⓐ	Fixed Point: Applies Λ to the state until further applications no longer modify the result. This final value is returned.
] ↦ [⊛]	List Wrap: Returns a list containing only a single item, the input.
^ ↦ Ⓐ	Restore: Restores state from the variable whose name is the symbol list of Λ.

` ↦ s	Image: Returns a string representation of the input.
o ↦	Loop: Applies Λ to the state indefinitely, does not return.
v ↦ (∗)	Save: Stores the state into the variable whose name is the symbol list of Λ then returns it without modification.
yλ ↦ [(A)]	Fixed Point List: Applies Λ to the state until further applications no longer modify the result. The final value is returned.
} ↦ (∗)	Set Wrap: Returns a set containing only a single item, the input.

Empty(Datatype) •*0*•. The empty type is the default starting type. It consists of only one value which represents a complete lack of information. From this value a number of generators can be produced.

Table 2: Operators for Empty •0•

OP ↦ RES	Description
'' ⊢ ∗S	String: Returns an empty String Constructor.
) ↦ ((∗))	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ ((∗)(A))	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ ((A)(∗))	Prepend: Inherited from Datatype on page 7.
W ↦ ((∗)(∗))	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [(∗)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
i ↦ (A)	Input: Returns an element of the type read from the source of Λ as inputted.
l ↦ QL	Quick Literal: Returns a type to pick a quick literal value.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (∗)	Set Wrap: Inherited from Datatype on page 8.
0 ↦ n	Zero: Returns zero as a natural number to begin specifying a natural number.

Integer(Datatype) •*i*•. The integer type represents a single unbounded integer n.

Table 3: Operators for Integer •i•

OP ↦ RES	Description
) ↦ ((∗))	Tuple Wrap: Inherited from Datatype on page 7.
+ ↦ n	Absolute Value: Returns the absolute value of n as a natural number.
, ↦ ((∗)(A))	Append: Inherited from Datatype on page 7.
- ↦ i	Negate: Returns $-n$ as an integer.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ ((A)(∗))	Prepend: Inherited from Datatype on page 7.
D ↦ i	Double: Returns $2n$ as an integer.
E ↦ B	Even: Returns true if n is even.
R ↦ [i]	Inclusive Range: Returns the list of integers from 0 to n inclusive.
W ↦ ((∗)(∗))	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [(∗)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
bλ ↦ [(A)]	Base Expand: Returns a base expansion for n, the base defined by .
o ↦	Loop: Inherited from Datatype on page 8.
r ↦ [i]	Exclusive Range: Returns the list of integers from 0 to n exclusive.
s ↦ i	Signum: Returns the signum of n.
v ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.

}	↦ (*)	Set Wrap: Inherited from Datatype on page 8.

Natural(Integer) •n•. The natural type represents a single unbound natural n.

Table 4: Operators for Natural •n•

OP	↦ RES	Description
!	↦ n	Factorial: Returns the factorial of n.
#	↦ n	Primorial: Returns the primorial of n.
)	↦ ((*))	Tuple Wrap: Inherited from Datatype on page 7.
+	↦ n	Add: Returns $n + x$ where x is .
,	↦ ((*)(A))	Append: Inherited from Datatype on page 7.
-	↦ i	Negate:
.	↦ (A)	Apply: Inherited from Datatype on page 7.
/	↦ Q	Inverse: Returns $\frac{1}{n}$ as a rational.
;	↦ ((A)(*))	Prepend: Inherited from Datatype on page 7.
D	↦ n	Double: Returns $2n$ as a natural number.
E	↦ B	Even: Returns true if n is even.
F	↦ n	Fibonacci: Returns the nth Fibonacci number.
H	↦ n∨Q	Halve: Returns $\frac{n}{2}$ as a natural number or as a rational.
L	↦ n	Lucas: Returns the nth Lucas number.
M	⊢ Mn	Math Mode: Returns a math mode operation on n.
O	↦ i	OEIS: Returns the nth element of OEIS Sequence.
P	↦ B	Prime: Returns true if n is a prime number.
R	↦ [i]	Inclusive Range: Inherited from Integer on page 8.
S	↦ n	Square: Returns n^2 as a natural number.
W	↦ ((*)(*))	Duplicate: Inherited from Datatype on page 7.
Y	↦ (A)	Fixed Point: Inherited from Datatype on page 7.
Z	↦	Zermelo: Returns the Zermelo ordinal representation of n.
]	↦ [(*)]	List Wrap: Inherited from Datatype on page 7.
^	↦ (A)	Restore: Inherited from Datatype on page 7.
`	↦ s	Image: Inherited from Datatype on page 8.
a	↦ n	Augment: Returns $n + 1$.
bλ	↦ [(A)]	Base Expand: Inherited from Integer on page 8.
d	↦ n∨i	Decrement: Returns $n - 1$ as a natural if possible, otherwise as an integer.
f	⊢ [n]	Factor: Returns the sorted list of all factors of n.
i	⊢ (nn)	Input: Read a natural number and return it and n in a pair (n first).
n	↦ n	Next Prime: Return the smallest prime number greater than n.
o	↦	Loop: Inherited from Datatype on page 8.
p	↦ n	Nth Prime: Returns the nth prime number (2 is the 0th).
r	↦ [i]	Exclusive Range: Inherited from Integer on page 8.
s	↦ i	Signum: Inherited from Integer on page 8.
v	↦ (*)	Save: Inherited from Datatype on page 8.
wλ	↦ S	Write: Write n in letters using the 'th representation.
yλ	↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
z	↦ B	Is Zero: Returns true if n is zero, false otherwise.
{	⊢	Von Neumann: Returns the Von Neumann ordinal representation of n.
}	↦ (*)	Set Wrap: Inherited from Datatype on page 8.
0	⊢ n	Times Ten: Returns $10n$ as a natural number.
1	⊢ n	Concatenate One: Returns $10n + 1$ as a natural number.
2	⊢ n	Concatenate Two: Returns $10n + 2$ as a natural number.
3	⊢ n	Concatenate Three: Returns $10n + 3$ as a natural number.
4	⊢ n	Concatenate Four: Returns $10n + 4$ as a natural number.
5	⊢ n	Concatenate Five: Returns $10n + 5$ as a natural number.
6	⊢ n	Concatenate Six: Returns $10n + 6$ as a natural number.
7	⊢ n	Concatenate Seven: Returns $10n + 7$ as a natural number.
8	⊢ n	Concatenate Eight: Returns $10n + 8$ as a natural number.
9	⊢ n	Concatenate Nine: Returns $10n + 9$ as a natural number.

Rational(Datatype) •*Q*•. The rational type represents a single rational value q.

Table 5: Operators for Rational •Q•

OP ↦ RES	Description
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (⊛A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A⊛)	Prepend: Inherited from Datatype on page 7.
W ↦ (⊛⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
d ↦ n	Denominator: Returns the denominator of the reduced form of q. The denominator of any q is positive.
n ↦ i	Numerator: Returns the numerator of the reduced form of q. If q is negative, so is its numerator.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ ⊛	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ⊛	Set Wrap: Inherited from Datatype on page 8.

Boolean(Datatype) •*B*•. The boolean type represents a single boolean value, either true or false.

Table 6: Operators for Boolean •B•

OP ↦ RES	Description
! ↦ B	Not: Returns the inverse of the value.
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (⊛A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A⊛)	Prepend: Inherited from Datatype on page 7.
?λ ↦ (A)	If: Returns the result of applying Λ to the empty type if the input is true, otherwise returns the input unchanged.
W ↦ (⊛⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ ⊛	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ⊛	Set Wrap: Inherited from Datatype on page 8.

Radigit(Natural) •*bn*•. This represents a digit in a given base. Its name is a portmanteau of radix and digit.

Table 7: Operators for Radigit •bn•

OP ↦ RES	Description
! ↦ n	Factorial: Inherited from Natural on page 9.
# ↦ n	Primorial: Inherited from Natural on page 9.
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
+ ↦ n	Add: Inherited from Natural on page 9.
, ↦ (⊛A)	Append: Inherited from Datatype on page 7.
- ↦ i	Negate: Inherited from Natural on page 9.
. ↦ A	Apply: Inherited from Datatype on page 7.

/ ↦ Q	Inverse: Inherited from Natural on page 9.
; ↦ (A *)	Prepend: Inherited from Datatype on page 7.
D ↦ n	Double: Inherited from Natural on page 9.
E ↦ B	Even: Inherited from Natural on page 9.
F ↦ n	Fibonacci: Inherited from Natural on page 9.
H ↦ n∨Q	Halve: Inherited from Natural on page 9.
L ↦ n	Lucas: Inherited from Natural on page 9.
M ⊢ Mn	Math Mode: Inherited from Natural on page 9.
O ↦ i	OEIS: Inherited from Natural on page 9.
P ↦ B	Prime: Inherited from Natural on page 9.
R ↦ [i]	Inclusive Range: Inherited from Integer on page 8.
S ↦ n	Square: Inherited from Natural on page 9.
W ↦ (* *)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
Z ↦	Zermelo: Inherited from Natural on page 9.
] ↦ [*]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
a ↦ n	Augment: Inherited from Natural on page 9.
bλ ↦ [A]	Base Expand: Inherited from Integer on page 8.
d ↦ n∨i	Decrement: Inherited from Natural on page 9.
f ⊢ [n]	Factor: Inherited from Natural on page 9.
i ⊢ (nn)	Input: Inherited from Natural on page 9.
n ↦ n	Next Prime: Inherited from Natural on page 9.
o ↦	Loop: Inherited from Datatype on page 8.
p ↦ n	Nth Prime: Inherited from Natural on page 9.
r ↦ [i]	Exclusive Range: Inherited from Integer on page 8.
s ↦ i	Signum: Inherited from Integer on page 8.
v ↦ *	Save: Inherited from Datatype on page 8.
wλ ↦ S	Write: Inherited from Natural on page 9.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
z ↦ B	Is Zero: Inherited from Natural on page 9.
{ ⊢	Von Neumann: Inherited from Natural on page 9.
} ↦ *	Set Wrap: Inherited from Datatype on page 8.
0 ⊢ n	Times Ten: Inherited from Natural on page 9.
1 ⊢ n	Concatenate One: Inherited from Natural on page 9.
2 ⊢ n	Concatenate Two: Inherited from Natural on page 9.
3 ⊢ n	Concatenate Three: Inherited from Natural on page 9.
4 ⊢ n	Concatenate Four: Inherited from Natural on page 9.
5 ⊢ n	Concatenate Five: Inherited from Natural on page 9.
6 ⊢ n	Concatenate Six: Inherited from Natural on page 9.
7 ⊢ n	Concatenate Seven: Inherited from Natural on page 9.
8 ⊢ n	Concatenate Eight: Inherited from Natural on page 9.
9 ⊢ n	Concatenate Nine: Inherited from Natural on page 9.

String(Datatype) •*S*•. This represents a string. Unlike most languages a string is a distinct entity from a list of characters, though conversion between the two is easy. 5b1t strings are fully unicode capable.

Table 8: Operators for String •S•

OP ↦ RES	Description
) ↦ (*)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (* A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A *)	Prepend: Inherited from Datatype on page 7.
B ⊢ [b]	To Bytes: Returns the input as a list of bytes in UTF-8 encoding.
Dλ ↦ i	Dictionary Index: Returns the index of the string in the th dictionary, or −1 if it does not appear in the dictionary (in any case).
L ⊢ [c]	To List: Returns the input as a list of characters.

R ⊢ S	Reverse: Returns the input reversed.
U ⊢ S	Upper Case: Returns the input in upper case.
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
aλ ↦ S	Append: Returns the image of Λ evaluated at an empty string constructor appended to the input.
dλ ↦ S	Dictionary Replace: Returns the first item of the th dictionary that has the shortest Damerau-Levenshtein distance to the input string.
l ⊢ n	Length: Returns the length of the string in characters.
o ↦	Loop: Inherited from Datatype on page 8.
pλ ↦ S	Prepend: Returns the image of Λ evaluated at an empty string constructor prepended to the input.
u ⊢ u	To URL: Returns the string as a URL.
v ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (∗)	Set Wrap: Inherited from Datatype on page 8.
~ ↦ S	Toggle Case: Returns the input with each character's case toggled.

Character(Datatype) •*C*•. This represents a single unicode character.

Table 9: Operators for Character •C•

OP ↦ RES	Description
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (∗A)	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ (A∗)	Prepend: Inherited from Datatype on page 7.
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (∗)	Set Wrap: Inherited from Datatype on page 8.

String Constructor(Datatype) •**S*•. This represents a string constructor. It is designed to quickly and efficiently construct any string desired. It does this by selecting from a number of encoding schemes and dictionaries. The image of a string constructor is the partially constructed string it contains.

Table 10: Operators for String Constructor •*S•

OP ↦ RES	Description
'' ⊢ S	Save: Returns the data in the string constructor as a string.
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (∗A)	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ (A∗)	Prepend: Inherited from Datatype on page 7.
Dλ ↦ ∗S	Dictionary: Sets the current dictionary to the th dictionary in the dictionary list (see page 20).
Wλ ↦ ∗S	Title Word: Loads the th word from the current dictionary (default English) in title case.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.

] ↦ [(*)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ (*)	Save: Inherited from Datatype on page 8.
wλ ↦ *S	Word: Loads the th word from the current dictionary (default English) in lower case.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.

Vector(Datatype) •*V*•. This represents any vector in any vector space. This type is not actually instantiable as it requires an underlying type. Nonetheless when instantiated by child types the following operations are available.

Table 11: Operators for Vector •V•

OP ↦ RES	Description
) ↦ ((*))	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ ((*)(A))	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ ((A)(*))	Prepend: Inherited from Datatype on page 7.
W ↦ ((*)(*))	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [(*)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ (*)	Save: Inherited from Datatype on page 8.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.

Quick Literal(Datatype) •*QL*•. This represents a type whose only purpose is to quickly provide some common literals. This is reminiscent of the many nilads present in most golfing languages.

Table 12: Operators for Quick Literal •QL•

OP ↦ RES	Description
) ↦ ((*))	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ ((*)(A))	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ ((A)(*))	Prepend: Inherited from Datatype on page 7.
A ⊢ S	Alphabet: Returns the string "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
B ⊢ S	Base Digits: Returns the string "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
C ⊢ S	Consonants: Returns the string "BCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz"
D ⊢ S	Digits: Returns the string "0123456789"

P ⊢ Q	Approximate Pi: Returns the rational number: 3208517945501666003498869248475748072433788353574911255830200151736103188805722685012705026506081695953513455858522187248050621440463818654734350551130531172632995135898519085037485600993649649837901211040733838943706637637771800748338263573876581647521468324080675775173734064352922352947732020883688896212134750237889823008201476996416819464335295656789519072488151724828507562064908712984130402269070362991479437491431733196628868122124284693300620007504071850855180341617755665994837033695336236881684510070000 / 1021302982051285186228530058081638356288973184353113784542588867022301381294952362353173069455054336962151267392300817622395581067396407038565323561796773894667172430002054609429747259228320634324230984436137563739089869540539996295763426584025926438277125188409344784890671412350181142991404644858213777779714325312006617574989320769501211582451893857378377782131880096471167630583191593337075260808852146105029831163523640417373112346060817540855435858638425158379734627102175969899866768643506427359579590359 which is a decent approximation of π (accurate to 1023 digits).
S ↦ S	Code Page: Returns the string: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/-|^&%<>()[]{}.,;:'"`~!@#$?_=\␣¶
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
O ↦	Loop: Inherited from Datatype on page 8.
V ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (∗)	Set Wrap: Inherited from Datatype on page 8.

Matrix(Datatype) •*M*•. This represents any matrix in any vector space. Like vectors, matrices are not actually instantiable as they require an underlying type. Nonetheless when instantiated by child types the following operations are available.

Table 13: Operators for Matrix •M•

OP ↦ RES	Description
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (∗A)	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ (A∗)	Prepend: Inherited from Datatype on page 7.
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
O ↦	Loop: Inherited from Datatype on page 8.
V ↦ (∗)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (∗)	Set Wrap: Inherited from Datatype on page 8.

Time(Datatype) •*T*•. This represents a moment in time, including date.

Table 14: Operators for Time •T•

OP ↦ RES	Description
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.

, ↦ (⊛ A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A ⊛)	Prepend: Inherited from Datatype on page 7.
W ↦ (⊛ ⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
O ↦	Loop: Inherited from Datatype on page 8.
V ↦ ⊛	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ⊛	Set Wrap: Inherited from Datatype on page 8.

Duration(Datatype) •*DT*•. This represents a duration in time, of any length.

Table 15: Operators for Duration •DT•

OP ↦ RES	Description
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (⊛ A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A ⊛)	Prepend: Inherited from Datatype on page 7.
W ↦ (⊛ ⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
O ↦	Loop: Inherited from Datatype on page 8.
V ↦ ⊛	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ⊛	Set Wrap: Inherited from Datatype on page 8.

URL(String) •*u*•. This represents a URL as specified by RFC 1738. It may not be valid, as any string may be converted to a URL.

Table 16: Operators for URL •u•

OP ↦ RES	Description
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (⊛ A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A ⊛)	Prepend: Inherited from Datatype on page 7.
? ⊢ B	Is Valid: Returns true if this represents a valid URL, false otherwise.
B ⊢ [b]	To Bytes: Inherited from String on page 11.
Dλ ↦ i	Dictionary Index: Inherited from String on page 11.
L ⊢ [c]	To List: Inherited from String on page 11.
R ⊢ S	Reverse: Inherited from String on page 12.
U ⊢ S	Upper Case: Inherited from String on page 12.
W ↦ (⊛ ⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ S	Image: Inherited from Datatype on page 8.
aλ ↦ S	Append: Inherited from String on page 12.
c ⊢ [S]	cURL: Returns the contents of the resource specified by the URL as a list of strings, one per line.
dλ ↦ S	Dictionary Replace: Inherited from String on page 12.

l ⊢ n	Length: Inherited from String on page 12.
o ↦	Loop: Inherited from Datatype on page 8.
pλ ↦ S	Prepend: Inherited from String on page 12.
u ⊢ u	To URL: Inherited from String on page 12.
v ↦ (*)	Save: Inherited from Datatype on page 8.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.
~ ↦ S	Toggle Case: Inherited from String on page 12.

List(Datatype) •[]•. The base list type is inherited by all lists of data. It is a heterogeneous list of values. Whenever a list is created or modified it checks what the nearest common ancestor of each element is. It represents itself as that base type. The most basic type is the "datatype" which shows up as no representation. This is analogous to the "Object" type in Java.

Table 17: Operators for List •[]•

OP ↦ RES	Description
) ↦ ((*))	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ ((*)(A))	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ ((A)(*))	Prepend: Inherited from Datatype on page 7.
F ⊢ [(1)]	Flatten Fully: Returns the list containing no containers. Each container is expanded into list elements.
S ↦ (*)	Shuffle: Returns the result of shuffling the list.
W ↦ ((*)(*))	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [(*)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ [(1)]	Flatten: Returns the list with one level of containerization removed.
i ↦ (A)	Index: Returns the th element of the list, indexed from 0.
l ↦ n	Length: Returns the number of elements in the list.
m ↦ [(1)]	Map: Returns the list created by applying Λ to each element in the list.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ (*)	Save: Inherited from Datatype on page 8.
yλ ↦ [(A)]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.

Integer List(List) •[i]•. This is a list consisting of only integers.

Table 18: Operators for Integer List •[i]•

OP ↦ RES	Description
) ↦ ((*))	Tuple Wrap: Inherited from Datatype on page 7.
+ ↦ i	Sum: Returns the sum of each element in the list.
, ↦ ((*)(A))	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
/ ↦ [i]	Sort Ascending: Returns the list sorted from smallest to largest.
; ↦ ((A)(*))	Prepend: Inherited from Datatype on page 7.
F ⊢ [(1)]	Flatten Fully: Inherited from List on page 16.
M ↦ i	Maximum: Returns the largest element in the list.
S ↦ (*)	Shuffle: Inherited from List on page 16.
W ↦ ((*)(*))	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
\ ↦ [i]	Sort Descending: Returns the list sorted from largest to smallest.
] ↦ [(*)]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.

` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ [1]	Flatten: Inherited from List on page 16.
i ↦ A	Index: Inherited from List on page 16.
l ↦ n	Length: Inherited from List on page 16.
m ↦ i	Minimum: Returns the smallest element in the list.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ *	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ *	Set Wrap: Inherited from Datatype on page 8.

Natural List(Integer List) •[n]•. This is a list consisting of only natural numbers.

Table 19: Operators for Natural List •[n]•

OP ↦ RES	Description
) ↦ (*)	Tuple Wrap: Inherited from Datatype on page 7.
+ ↦ i	Sum: Inherited from Integer List on page 16.
, ↦ (*A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
/ ↦ [i]	Sort Ascending: Inherited from Integer List on page 16.
; ↦ (A*)	Prepend: Inherited from Datatype on page 7.
F ⊢ [1]	Flatten Fully: Inherited from List on page 16.
M ↦ i	Maximum: Inherited from Integer List on page 16.
P ↦ n	Deprime: Returns the number created by raising each prime to the value in the list at its index and multiplying them together.
S ↦ *	Shuffle: Inherited from List on page 16.
W ↦ (**)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
\ ↦ [i]	Sort Descending: Inherited from Integer List on page 16.
] ↦ [*]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ [1]	Flatten: Inherited from List on page 16.
i ↦ A	Index: Inherited from List on page 16.
l ↦ n	Length: Inherited from List on page 16.
m ↦ i	Minimum: Inherited from Integer List on page 17.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ *	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ *	Set Wrap: Inherited from Datatype on page 8.

Boolean List(List) •[B]•. This is a list consisting of only boolean values.

Table 20: Operators for Boolean List •[B]•

OP ↦ RES	Description
& ↦ B	All: Returns true if every element in the list is true
) ↦ (*)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (*A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A*)	Prepend: Inherited from Datatype on page 7.
F ⊢ [1]	Flatten Fully: Inherited from List on page 16.
S ↦ *	Shuffle: Inherited from List on page 16.
W ↦ (**)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [*]	List Wrap: Inherited from Datatype on page 7.
^ ↦ B	Parity: Returns true if there an odd number of true elements in the list.

` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ [1]	Flatten: Inherited from List on page 16.
i ↦ A	Index: Inherited from List on page 16.
l ↦ n	Length: Inherited from List on page 16.
m ↦ [1]	Map: Inherited from List on page 16.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ ∗	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
| ↦ B	Any: Returns true if any element in the list is true.
} ↦ ∗	Set Wrap: Inherited from Datatype on page 8.

Character List(List) •*[C]*•. This is a list consisting of only characters. This represents the same data as a string, but in a different form with different operations available.

Table 21: Operators for Character List •[C]•

OP ↦ RES	Description
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (∗A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A∗)	Prepend: Inherited from Datatype on page 7.
F ⊢ [1]	Flatten Fully: Inherited from List on page 16.
S ↦ ∗	Shuffle: Inherited from List on page 16.
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ [1]	Flatten: Inherited from List on page 16.
i ↦ A	Index: Inherited from List on page 16.
l ↦ n	Length: Inherited from List on page 16.
m ↦ [1]	Map: Inherited from List on page 16.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ ∗	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ∗	Set Wrap: Inherited from Datatype on page 8.

Tuple(Datatype) •*()*•. The base tuple type is inherited by all tuples of data.

Table 22: Operators for Tuple •()•

OP ↦ RES	Description
) ↦ (∗)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (∗A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A∗)	Prepend: Inherited from Datatype on page 7.
E ⊢ (1)	Ends: Returns the pair consisting of the first and last elements of the tuple.
F ⊢ (1)	Flatten Fully: Returns the tuple containing no containers. Each container is expanded into tuple elements.
T ⊢ (1)	Triple: Returns the triple consisting of the first, middle (rounding up) and last elements of the tuple.
W ↦ (∗∗)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [∗]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.

f ⊢ (1)	Flatten: Returns the tuple with one level of containerization removed.
o ↦	Loop: Inherited from Datatype on page 8.
t ⊢ (1)	Triple: Returns the triple consisting of the first, middle (rounding down) and last elements of the tuple.
v ↦ (*)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.
0 ↦ (A)	Nth: Returns the nth value of the tuple, indexed from 0.
1 ↦ (1)	On First: Applies Λ to the first value of the tuple.
2 ↦ (1)	On Second: Applies Λ to the second value of the tuple.
3 ↦ (1)	On Third: Applies Λ to the third value of the tuple.
4 ↦ (1)	On Fourth: Applies Λ to the fourth value of the tuple.
5 ↦ (1)	On Fifth: Applies Λ to the fifth value of the tuple.
6 ↦ (1)	On Sixth: Applies Λ to the sixth value of the tuple.
7 ↦ (1)	On Seventh: Applies Λ to the seventh value of the tuple.
8 ↦ (1)	On Eighth: Applies Λ to the eighth value of the tuple.
9 ↦ (1)	On Ninth: Applies Λ to the ninth value of the tuple.

Integer Pair(Tuple) •*(ii)*•. This is a tuple consisting of two integers, x and y.

Table 23: Operators for Integer Pair •(ii)•

OP ↦ RES	Description
) ↦ (*)	Tuple Wrap: Inherited from Datatype on page 7.
+ ↦ i	Add: Returns $x + y$ as an integer.
, ↦ (*A)	Append: Inherited from Datatype on page 7.
. ↦ (A)	Apply: Inherited from Datatype on page 7.
; ↦ (A*)	Prepend: Inherited from Datatype on page 7.
E ⊢ (1)	Ends: Inherited from Tuple on page 18.
F ⊢ (1)	Flatten Fully: Inherited from Tuple on page 18.
T ⊢ (1)	Triple: Inherited from Tuple on page 18.
W ↦ (**)	Duplicate: Inherited from Datatype on page 7.
Y ↦ (A)	Fixed Point: Inherited from Datatype on page 7.
] ↦ [*]	List Wrap: Inherited from Datatype on page 7.
^ ↦ (A)	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ (1)	Flatten: Inherited from Tuple on page 19.
o ↦	Loop: Inherited from Datatype on page 8.
t ⊢ (1)	Triple: Inherited from Tuple on page 19.
v ↦ (*)	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ (*)	Set Wrap: Inherited from Datatype on page 8.
0 ↦ (A)	Nth: Inherited from Tuple on page 19.
1 ↦ (1)	On First: Inherited from Tuple on page 19.
2 ↦ (1)	On Second: Inherited from Tuple on page 19.
3 ↦ (1)	On Third: Inherited from Tuple on page 19.
4 ↦ (1)	On Fourth: Inherited from Tuple on page 19.
5 ↦ (1)	On Fifth: Inherited from Tuple on page 19.
6 ↦ (1)	On Sixth: Inherited from Tuple on page 19.
7 ↦ (1)	On Seventh: Inherited from Tuple on page 19.
8 ↦ (1)	On Eighth: Inherited from Tuple on page 19.
9 ↦ (1)	On Ninth: Inherited from Tuple on page 19.

Set(Datatype) •*{}*•. This represents a set.

Table 24: Operators for Set •{}•

OP ↦ RES	Description
) ↦ (⊛)	Tuple Wrap: Inherited from Datatype on page 7.
, ↦ (⊛A)	Append: Inherited from Datatype on page 7.
. ↦ A	Apply: Inherited from Datatype on page 7.
; ↦ (A⊛)	Prepend: Inherited from Datatype on page 7.
F ⊢ 1	Flatten Fully: Returns the set containing no containers. Each container is expanded into set elements.
W ↦ (⊛⊛)	Duplicate: Inherited from Datatype on page 7.
Y ↦ A	Fixed Point: Inherited from Datatype on page 7.
] ↦ [⊛]	List Wrap: Inherited from Datatype on page 7.
^ ↦ A	Restore: Inherited from Datatype on page 7.
` ↦ s	Image: Inherited from Datatype on page 8.
f ⊢ 1	Flatten: Returns the set with one level of containerization removed.
o ↦	Loop: Inherited from Datatype on page 8.
v ↦ ⊛	Save: Inherited from Datatype on page 8.
yλ ↦ [A]	Fixed Point List: Inherited from Datatype on page 8.
} ↦ ⊛	Set Wrap: Inherited from Datatype on page 8.

Execution

Installation

The program can be installed by simply running the `make && make install` idiom. This uses the environment variable `PREFIX` as the installation directory if not otherwise specified.

Driver

The `5b1t` executable is the main driver for 5b1t code. It has a basic `--help` option, but its behaviour will be more precisely explained.

The driver takes the input file as a positional argument. If no such argument is present, then it reads it from standard input.

By default, the inputted file is compiled and run. This can cause a subtle problem if the input is provided by standard input instead of a positional file. In this case the compilation phase will have already consumed all of the available standard input, leaving nothing for the compiled program to consume. This may not be an issue for programs which take no input, but for those that do it can lead to unusual behaviour.

The default behaviour of running the code can be modified via command-line arguments. The `-c` option causes the driver to only compile the source (by default creating an executable named "a.out"). The `-d` option causes the driver to decode the input from 5b1t encoding to ASCII. The `-e` option causes the driver to encode the input from ASCII to 5b1t encoding. Both the `-d` and `-e` modes of operation dump their results to standard output, ignoring any provided output file. The `-c` options output executable can be modified by specifying an alternative using the `-o` option. If the `-o` option is present, then it is assumed that the desired behaviour is compilation, thus `-c` is not necessary.

Also by default, the inputted file is assumed to be encoded as per the above encoding. If the `-a` switch is passed then it is assumed to be ASCII encoded instead, treating tab characters as line comment initializers.

The initial type on which the program will operate (and which will determine how the initial object is parsed from input) can be specified using the `-i` option. If not specified the initial type defaults to the nothing type.

For debugging purposes the `-g` option can be specified, causing the resulting execution to print partial results to standard error. Similarly the `-s` option provides the capability to save the generated Ada source code to a known file for inspection.

The random number generator used by the code can be pre-seeded with an integer using the `-r` option. A seed of 0 (the default) will use a dynamic seed.

Dictionaries

Many dictionaries are accessible from within 5b1t programs. Most of them are based on the dictionaries at `https://ftp.gnu.org/gnu/aspell/dict/0index.html`.

These are found in the dictionary files in the share directory once they are downloaded.

The indices of the dictionaries, and their descriptions, are as follows:

ID#	Name	Description

0	English	The English words as conglomerated from aspell dictionaries for multiple dialects.
1	Afrikaans	The Afrikaans words as conglomerated from the aspell dictionary.
2	Amharix	The Amharic words as conglomerated from the aspell dictionary.
3	Arabic	The Arabic words as conglomerated from the aspell dictionary.
4	Asturian	The Asturian words as conglomerated from the aspell dictionary.
5	Azerbaijani	The Azerbaijani words as conglomerated from the aspell dictionary.
6	Belarusian	The Belarusian words as conglomerated from the aspell dictionary.
7	Bulgarian	The Bulgarian words as conglomerated from the aspell dictionary.
8	Bengali	The Bengali words as conglomerated from the aspell dictionary.
9	Breton	The Breton words as conglomerated from the aspell dictionary.
10	Catalan	The Catalan words as conglomerated from the aspell dictionary.
11	Czech	The Czech words as conglomerated from the aspell dictionary.
12	Kashubian	The Kashubian words as conglomerated from the aspell dictionary.
13	Welsh	The Welsh words as conglomerated from the aspell dictionary.
14	Danish	The Danish words as conglomerated from the aspell dictionary.
15	German	The German words as conglomerated from the aspell dictionary.
16	German - Old Spelling	The German words in old spelling as conglomerated from the aspell dictionary.
17	Greek	The Greek words as conglomerated from the aspell dictionary.
18	Esperanto	The Esperanto words as conglomerated from the aspell dictionary.
19	Spanish	The Spanish words as conglomerated from the aspell dictionary.
20	Estonian	The Estonian words as conglomerated from the aspell dictionary.
21	Persian	The Persian words as conglomerated from the aspell dictionary.
22	Finnish	The Finnish words as conglomerated from the aspell dictionary.
23	Faroese	The Faroese words as conglomerated from the aspell dictionary.
24	French	The French words as conglomerated from the aspell dictionary.
25	Frisian	The Frisian words as conglomerated from the aspell dictionary.
26	Irish	The Irish words as conglomerated from the aspell dictionary.
27	Scottish Gaelic	The Scottish Gaelic words as conglomerated from the aspell dictionary.
28	Galician	The Galician words as conglomerated from the aspell dictionary.
29	Ancient Greek	The Ancient Greek words as conglomerated from the aspell dictionary.
30	Gujarati	The Gujarati words as conglomerated from the aspell dictionary.
31	Manx Gaelic	The Manx Gaelic words as conglomerated from the aspell dictionary.
32	Hebrew	The Hebrew words as conglomerated from the aspell dictionary.
33	Hindi	The Hindi words as conglomerated from the aspell dictionary.
34	Hiligaynon	The Hiligaynon words as conglomerated from the aspell dictionary.
35	Croatian	The Croatian words as conglomerated from the aspell dictionary.
36	Upper Sorbian	The Upper Sorbian words as conglomerated from the aspell dictionary.
37	Hungarian	The Hungarian words as conglomerated from the aspell dictionary.
38	Huastec	The Huastec words as conglomerated from the aspell dictionary.
39	Armenian	The Armenian words as conglomerated from the aspell dictionary.
40	Interlingua	The Interlingua words as conglomerated from the aspell dictionary.
41	Indonesian	The Indonesian words as conglomerated from the aspell dictionary.
42	Icelandic	The Icelandic words as conglomerated from the aspell dictionary.
43	Italian	The Italian words as conglomerated from the aspell dictionary.
44	Kannada	The Kannada words as conglomerated from the aspell dictionary.
45	Kurdi	The Kurdi words as conglomerated from the aspell dictionary.
46	Kirghiz	The Kirghiz words as conglomerated from the aspell dictionary.
47	Latin	The Latin words as conglomerated from the aspell dictionary.
48	Lithuanian	The Lithuanian words as conglomerated from the aspell dictionary.
49	Latvian	The Latvian words as conglomerated from the aspell dictionary.
50	Malagasy	The Malagasy words as conglomerated from the aspell dictionary.
51	Maori	The Maori words as conglomerated from the aspell dictionary.
52	Macedonian	The Macedonian words as conglomerated from the aspell dictionary.
53	Malayalam	The Malayalam words as conglomerated from the aspell dictionary.
54	Mongolian	The Mongolian words as conglomerated from the aspell dictionary.
55	Marathi	The Marathi words as conglomerated from the aspell dictionary.
56	Malay	The Malay words as conglomerated from the aspell dictionary.
57	Maltese	The Maltese words as conglomerated from the aspell dictionary.
58	Norwegian Bokmal	The Norwegian Bokmal words as conglomerated from the aspell dictionary.
59	Low Saxon	The Low Saxon words as conglomerated from the aspell dictionary.
60	Dutch	The Dutch words as conglomerated from the aspell dictionary.

61	Norwegian Nynorsk	The Norwegian Nynorsk words as conglomerated from the aspell dictionary.
62	Chichewa	The Chichewa words as conglomerated from the aspell dictionary.
63	Oriya	The Oriya words as conglomerated from the aspell dictionary.
64	Punjabi	The Punjabi words as conglomerated from the aspell dictionary.
65	Polish	The Polish words as conglomerated from the aspell dictionary.
66	Brazilian Portuguese	The Brazilian Portuguese words as conglomerated from the aspell dictionary.
67	Portuguese	The Portuguese words as conglomerated from the aspell dictionary.
68	Quechua	The Quechua words as conglomerated from the aspell dictionary.
69	Romanian	The Romanian words as conglomerated from the aspell dictionary.
70	Russian	The Russian words as conglomerated from the aspell dictionary.
71	Kinyarwanda	The Kinyarwanda words as conglomerated from the aspell dictionary.
72	Sardinian	The Sardinian words as conglomerated from the aspell dictionary.
73	Slovak	The Slovak words as conglomerated from the aspell dictionary.
74	Slovenian	The Slovenian words as conglomerated from the aspell dictionary.
75	Serbian	The Serbian words as conglomerated from the aspell dictionary.
76	Swedish	The Swedish words as conglomerated from the aspell dictionary.
77	Swahili	The Swahili words as conglomerated from the aspell dictionary.
78	Tamil	The Tamil words as conglomerated from the aspell dictionary.
79	Telugu	The Telugu words as conglomerated from the aspell dictionary.
80	Tetum	The Tetum words as conglomerated from the aspell dictionary.
81	Turkmen	The Turkmen words as conglomerated from the aspell dictionary.
82	Tagalog	The Tagalog words as conglomerated from the aspell dictionary.
83	Setswana	The Setswana words as conglomerated from the aspell dictionary.
84	Turkish	The Turkish words as conglomerated from the aspell dictionary.
85	Ukrainian	The Ukrainian words as conglomerated from the aspell dictionary.
86	Uzbek	The Uzbek words as conglomerated from the aspell dictionary.
87	Vietnamese	The Vietnamese words as conglomerated from the aspell dictionary.
88	Walloon	The Walloon words as conglomerated from the aspell dictionary.
89	Yiddish	The Yiddish words as conglomerated from the aspell dictionary.
90	Zulu	The Zulu words as conglomerated from the aspell dictionary.

Table 25: 5b1t Dictionaries

FAQ

What is the 5b1t logo?

The logo is a simplified vector rendition of a *Mola* fry. The *Mola* or sunfish is a fairly well-known fish. It is the largest known bony fish at over to 3 m long and weighing over 2 t. However, being a *tetraodontiform* closely related to boxfishes and puffer fish, it has very tiny fry (fish babies) measuring only 2.5 mm long and weighing less than a gram. These fry look a lot like a tiny puffer fish, showing the *Mola*'s ancestry.

A *Mola* fry was chosen as the 5b1t logo due to the large growth it undergoes through its lifetime. The sunfish has a claim to be the animal with the largest growth over its lifetime, growing over 60 million times its size at birth by the time it is an adult. Similarly 5b1t programs expand tiny source into massive executables (the empty program expands to over 5 mb of executable) with enormous potential.

Despite the myth that sunfish diets consist mostly of jellyfish, 5b1t is intended to be a friendly competitor to Dennis Mitchell's Jelly and not a predator thereof.