Kamila Szewczyk

MalbolgeLISP

DESIGN AND IMPLEMENTATION OF THE MOST COMPLEX MALBOLGE UNSHACKLED PROGRAM TO DATE

Fall 2020 - Fall 2021

Abstract

MalbolgeLISP is the most complex Malbolge Unshackled program to date (2020, 2021). Unlike other Malbolge programs generated by different toolchains (for instance, LAL, HAL or the "pseudo-instruction" language developed by the Nagoya university), MalbolgeLISP can be used to express complex computations (like folds, monads, efficient scans, iteration and point-free programming), while being able to run within reasonable time and resource constrains on mid-end personal computers. The project aims to research not the cryptanalysis aspect of Malbolge, but its suitability for complex appliances, which could be useful for cryptography and intellectual property protection, and it would certainly raise the bar for future Malbolge programs while exploring functional and array programming possibilities using inherently imperative and polymorphism-oriented Malbolge code.

Foreword

MalbolgeLISP is currently one of my most popular projects. It has recently gained traction on sites like Reddit[1], HackerNews[2] GitHub[3], *surprisingly Turing-complete*[4] and many others. During the fall of 2020, as well as the summer and fall of 2021, I spent my time optimising my toolkit, working on the Malbolge interpreter and MalbolgeLISP, troubleshooting performance problems, implementing new features, and testing them.

In 2021, two new versions of MalbolgeLISP were released - v1.1 and v1.2, the first one featuring performance optimisations and a few additional features, and the second one featuring many functional devices and a fast Malbolge interpreter. Across the releases, the codebase became harder to test (lack of testing hardware, limited amounts of test programs) and harder to work with (increasingly long compilation times, larger code size, more complicated compilation pipeline, since I introduced new features to my toolchain while developing MalbolgeLISP, etc...). In particular, I'd like to thank Github user Matt8898[5] for providing me test cases and feedback.

During the course of this project, I've been drawing inspiration from APL (Dyalog APL[6] in particular) and Haskell[7]. Their influence is evident judging by the function set and syntax extensions to traditional Lisp, while the APL influence is additionally emphasised by providing equivalent expressions to MalbolgeLISP built-in operations or example expressions, as well as explaining the core concepts of MalbolgeLISP using APL as a tool of thought.

[1] `https://www.reddit.com/r/lisp/comments/oxtpnn/kspalaiologosmalbolgelisp_a_lightweight_150mb/`

[2] `https://news.ycombinator.com/item?id=28048072`

[3] `https://github.com/kspalaiologos/malbolge-lisp`, with around 60'000 visits within the first few days of publishing and keeping the profile of 60 unique visit each day after a few weeks.

[4] `https://www.gwern.net/Turing-complete`

[5] `https://github.com/Matt8898/`

[6] `https://www.dyalog.com/`

[7] `https://www.haskell.org/`

Contents

Glossary

In this book, the following assumptions are made:

- `*X` is the value *pointed* by `X` - for instance, if the 3rd word of memory is `152` and `X` is `3`, then `*X` is `152`.
- `X[Y]` is equivalent to `*(X + Y)`. Because addition is commutative, `X[Y]` is equivalent to `Y[X]`.
- *Folding* a list `ω` with a function `αα` is equivalent to putting `αα` between every element of `ω`, also given the identity element of `αα`. One could assume binding to the left (`(ω[1] αα ω[2]) αα ω[3]`), or to the right (`ω[1] αα (ω[2] αα ω[3])`), hence the names *fold-right* and *fold-left*. Unlike reduction, the result of *folding* an empty array is defined and equal to the identity element.
- *Scanning* a list `ω` with a function `αα` is equivalent to mapping a fold over prefixes of a list. This definition of a scan has a $O(n^2)$ complexity regardless of the fold type. A more efficient, $O(n)$ definition of a *scan-left* exists (fold with intermediate steps).
- *Outer product* of function `αα` (`∘.αα`) and arrays `α` and `ω` is the application of `αα` between every pair of elements from `α` and `ω`. The resulting list has a depth given by `ρ,⍤ρ`.
- *Inner product* of functions `αα`, `ωω` and lists `α` and `ω` (`αα.ωω`) is equivalent to folding the list obtained by putting `ωω` between corresponding pairs of `α` and `ω` with `αα`.
- A *higher-order function* is a function that takes another function as its argument.
- A *lambda expression* (in the context of MalbolgeLISP) is an anonymous function with static (lexical) scoping, meaning that it can see all the variables bound by its lexical ancestor, unlike to dynamic scoping implemented by older LISPs where a function can see all the variables bound by its caller(s). In both cases, the first bound variable found is used, allowing to shadow variables.
- A *dyad* is a two argument function.
- A *monad* is generally a single argument function. The book also uses it in a context of an abstract data constructor implementing the *bind* and *unit* functions.
- *Replicating* a list `ω` according to list `α` is copying each element of `ω` a given number of times (specified by the corresponding element of `α` or assumed to be zero). In MalbolgeLISP, `α` can be a scalar, in which case the contents of `ω` are catenated to each other `α` times. If `ω` is also a scalar, it's repeated `α` times to form a list.
- *Filtering* a list `ω` with function `αα` is equivalent to mapping `αα` on every element of `ω` (assuming `αα` returns a boolean value, i.e. `0` or `1`), and replicating `ω` with the result of the mapping (`ω/⍤αα⍤ω`).
- *Rotating* a list `ω` by `α` elements is equivalent to dropping `α` elements from `ω`, and joining them with `ω` (dyadic `⍞`, or more illustratively, `(↓,↑)`).
- *Zipping* two arrays is the act of forming a list of pairs via the juxtaposition of corresponding elements from the given arrays. Alternatively, the pairs can be processed by a functor (*zipwith*).
- *Flattening* a list if the act of decreasing the list's depth by one level, enlisting all of the elements from the topmost sublists into a resultant list.

- *Mapping* a function `αα` over a list `ω` is the act of processing each element of `ω` with the function `αα` to produce a resultant list of equivalent length.
- *Partial application* is defined as fixing a number of arguments to a function, yielding an anonymous function of smaller arity.
- *Iteration* of a function over an argument is defined as evaluating the function over its result starting with the initial argument until a condition is satisfied. Sometimes the condition is numeric (`iterateN` MalbolgeLISP word), or is defined as a dyadic function between the previous and current result (just `iterate`). A fixed point combinator could be implemented using partial application of deep equality (MalbolgeLISP: `bind iterate =`, APL: `⍣≡`). Iteration is faster and consumes less resources than recursion, assuming no tail call optimisation.
- *Function composition* (or `atop`, as MalbolgeLISP calls it) is taking an arbitrary amount of functions where each function operates on the results of the function before, except the last function, which gets all the arguments passed to the anonymous function yielded by composition - in MalbolgeLISP, `((atop f g h) a b c)` ⟺ `(f (g (h a b c)))`.
- *Selfie* is a higher order function that duplicates the argument to the function it takes (i.e. `(bind selfie *)`, equivalent to `×⍨`, computes the square of its argument).
- *Commute* swaps the order of two arguments to a function. `α f ω` ⟺ `ω f⍨ α`.
- `CON0` is a 10-trit number containing only zeroes.
- `CON1` is a 10-trit number containing only ones.
- `CON2` is a 10-trit number containing only twos.

Chapter 1

Malbolge and Malbolge Unshackled

Malbolge, invented by Ben Olmstead in 1998, is an esoteric programming language designed to be as difficult to program in as possible. A few key characteristics of it follow:

- A virtual machine based on the ternary system.
- Von Neumann architecture - code and data aren't separated from each other.
- Memory is in a deterministic state, yet it isn't zeroed. Instead, it's filled with a sequence produced as a derivative of *crazy operation*.
- Each machine word is 10 trits wide (fixed and relatively small rotation width; small amount of addressable memory - less than 65K).
- Each register and memory cell holds a single machine word.
- Only three registers:
  - `A` - accumulator
  - `C` - *code* (instruction) pointer
  - `D` - data pointer
- Eight basic operations.
- The opcode executed depends on its position in the file and encrypted after being executed (thus, provoking the thought of instruction cycles - what if on some position, a certain instruction produces a looping cycle that might have a purpose?).
- The instruction set includes a jump instruction, a halt instruction, I/O routines, no-operation (which is used almost exclusively in NOP sleds), pointer dereference and two operations on data.
- The data can only be modified via tritwise rotation (single trit at a time) or an unintuitive[1] *crazy* operation.
- Early toy programs made in Malbolge[2] [3] were made either via bruteforce (first *"Hello, world"* program) or inefficient tooling. Some programs were made by hand, like the following `cat` program that doesn't terminate[4]:

```
(=BA#9"=<;:3y7x54-21q/p-,+*)"!h%B0/.
~P<<:(8&66#"!~}|{zyxwvugJ%
```

[1] according to Esolangs wiki, `https://esolangs.org/wiki/Malbolge`

[2] `https://esolangs.org/wiki/Malbolge_programming`

[3] `http://www.lscheffer.com/malbolge.shtml`

[4] `https://esolangs.org/wiki/Malbolge#Cat`

		A		
		0	**1**	**2**
[D]	**0**	1	0	0
	1	1	0	2
	2	2	2	1

Table 1.2: Malbolge's crazy operation executed on single trits.

opcode	normalised	operation
4	`i`	`C = *D`
5	`<`	`putchar(A % 256)`
23	`/`	`A = getchar()`
39	`*`	`A = *D = rot_r(*D)`
40	`j`	`D = *D`
62	`p`	`A = *D = crzop(A, *D)`
68	`o`	`/* no-operation */`
81	`v`	`exit(0)`

Table 1.1: Malbolge's 8 basic operations

Since Malbolge can address only 3^{10} memory cells, it's definitely **not** Turing-complete. Neither are extensions to it, like *Malbolge20*[5], since the addressable memory is still bounded. MalbolgeLISP is a Malbolge Unshackled program (which doesn't depend on a fixed rotation width, but using a rotwidth loop, it makes sure that the rotation width is wide enough).

To outline the most important differences between Malbolge Unshackled and Malbolge:

- The rotation width is chosen randomly by the interpreter.
- Malbolge Unshackled lets the width of rotation be variable, which grows with the values in the D register, and since the initial rotation width is unknown, one would have to probe it (otherwise `*` returns unpredictable results).
- Malbolge Unshackled's print instruction takes unicode codepoints.
- If the rotation width is unknown, it's theoretically impossible to load values larger than $3^4 - 1$, except values starting with a `1` trit.

1.1. Striving for a fast interpreter

MalbolgeLISP is available for download on my website[6], or from the GitHub repository[7]. It's bundled with a performant Malbolge Unshackled interpreter with a fixed rotation width of 19 trits, confusingly called *fast20*. MalbolgeLISP v1.1 used to be bundled with a different version of fast20, which is currently obsolete because of its unsatisfactory performance and excessive resource consumption. It could be argued that fast20 is not a valid Malbolge Unshackled interpreter (the rotation behavior diverges from the Malbolge Unshackled specification), yet MalbolgeLISP is still theoretically usable on a dynamic rotation width interpreter. Such an interpreter has not been provided, since they tend to be very slow, and no extended tests have been made - because of the internal tooling structure however, MalbolgeLISP still manages to achieve 100% coverage.

The idea of fixing the rotation width was already employed in the past, although all implementations picked the rotation width 20. The rotation width of 19 trits is marginally faster at an insignificant cost, hence it's used in fast20.

[5] `https://www.trs.css.i.nagoya-u.ac.jp/projects/Malbolge/`
[6] `https://kamila.akagi.moe/malbolgelisp-v1.2/`
[7] `https://github.com/kspalaiologos/malbolge-lisp`

1.1.1. Data representation

Ørjan Johansen[8] uses the following data structures in his Malbolge Unshackled interpreter:

```
-- The memory structure, a combined trie and linked list
data MemNode = MemNode {
    nodes :: MemNodes, next :: MemNode,
    value :: IORef Value, modClass :: Int, width :: Int}
type MemNodes = Trit -> MemNode
data Value = OffsetV Trit Integer | ListV [Trit] deriving (Show)
data Trit = T0 | T1 | T2 deriving (Enum, Show, Eq)

type UMonad = StateT UState IO
data UState = UState {
    other :: OtherState,
    a :: Value,
    c :: MemNode,
    d :: MemNode }
data OtherState = Other { -- Things that change rarely, if at all
    memory :: MemNode,
    rotWidth :: Int,
    maxWidth :: Int,
    growthPolicy :: Int -> UMonad OtherState }
```

It is apparent that the memory isn't implemented in a particularly efficient manner (nodes form a list, machine word is a list). Rotation is implemented as a recursive operation and crazy operation is implemented as mapping of a trit-wise `op` on each trit:

```
rotate width val = ListV $ compressList $ rot w t r where
    w = if width >= 1 then width else bug "..."
    ListV (t:r') = vToList val
    r = case r' of
        []  -> [t]
        _   -> r'
    rot 1 t r       = t : r
    rot w t [l]     = l : rot (w-1) t [l]
    rot w t (t':l)  = t' : rot (w-1) t l

opValue v1 v2 = ListV $ compressList $ opv l1 l2 where
    opv [t] l = map (t `op`) l
    opv l [t] = map (`op` t) l
    opv (t1:r1) (t2:r2) = (t1 `op` t2):opv r1 r2
    opv _ _ = bug "..."
    ListV l1 = vToList v1
    ListV l2 = vToList v2

op T0 T0 = T1;      op T1 T0 = T0;      op T2 T0 = T0
op T0 T1 = T1;      op T1 T1 = T0;      op T2 T1 = T2
op T0 T2 = T2;      op T1 T2 = T2;      op T2 T2 = T1
```

A model with superior performance characteristics would involve fixing the rotation width to some value, thus making the memory a contiguous vector, making rotation and crazy operation $O(1)$. This model has been employed in early versions fast20 (derived from Matthias Lutter's public domain code[9]).

[8] http://oerjan.nvg.org/esoteric/Unshackled.hs
[9] https://lutter.cc/unshackled/Unshackled-20.c

```c
typedef struct Word {
    #ifndef MEMORY
        unsigned int area;
        unsigned int high;
        unsigned int low;
    #else
        unsigned char area;
        unsigned short high;
        unsigned short low;
    #endif
} Word;

static inline uint16_t crazy_low(uint16_t a, uint16_t d) {
    const uint16_t crz[] = { 1, 0, 0, 1, 0, 2, 2, 2, 1 };
    uint16_t result = 0; uint16_t k = 1;
    for(char pos = 0; pos < 10; pos++) {
        result += k * crz[(a % 3) + 3 * (d % 3)];
        a /= 3; d /= 3; k *= 3;
    }
    return result;
}

static inline Word zero() {
    Word result = {0, 0, 0};
    return result;
}

static inline Word increment(Word d) {
    d.low++;
    if (d.low >= 59049) {
        d.low = 0;
        d.high++;
    }
    return d;
}

static inline Word decrement(Word d) {
    if (d.low == 0) {
        d.low = 59048;
        d.high--;
    } else
        d.low--;
    return d;
}

static inline Word crazy(Word a, Word d) {
    Word output;
    unsigned int crz[] = {1,0,0,1,0,2,2,2,1};
    output.area = crz[a.area+3*d.area];
    output.high = crazy_low(a.high, d.high);
    output.low = crazy_low(a.low, d.low);
    return output;
}
```

```
static inline Word rotate_r(Word d) {
    unsigned int carry_h = d.high % 3;
    unsigned int carry_l = d.low % 3;
    d.high = 19683 * carry_l + ((unsigned int) d.high) / 3;
    d.low = 19683 * carry_h + ((unsigned int) d.low) / 3;
    return d;
}
```

The memory management scheme of this interpreter is also unsatisfactory. A single memory cell takes three `unsigned int`s worth of memory (on the testing machine used while developing fast20, `sizeof(unsigned int) == 4`), which is almost three times less memory efficient as it could be (since $2^{32} > 3^{20}$).

Since the `high` and `low` fields store 10 trits, they can be turned into `unsigned short`s. The `area` field stores a single trit, so it can be made an `unsigned char`. This is what happens when `MEMORY` is defined, but in this case, unaligned accesses worsen the performance of the program (25s vs 30s for `(! 6)`), which is still not satisfactory, since MalbolgeLISP v1.2's interpreter takes only 8s to execute this expression (12s if instead of `unsigned int` memory cells, `unsigned long` memory cells are forced), meaning that it's more efficient than old fast20 in any configuration.

The most effective memory management scheme for a Malbolge Unshackled interpreter of fixed rotation width considered so far involves treating memory cells as unsigned 32-bit integers, without distinction between the higher or lower bits.

```
#define u8          uint8_t
#define u32         uint32_t
#define C           const
#define P           static
#define _(a...)     {return({a;});}
#define F_(n,a...) for(int i=0;i<n;i++){a;}
#define INLINE P inline __attribute__((always_inline))

typedef u32 W;
#define SZ 19
#define END 1162261467ULL

P C u8 crz[] = {
    1,0,0,9,
    1,0,2,9,
    2,2,1
}, crz2[] = {
    4,3,3,1,0,0,1,0,0,9,9,9,9,9,9,9,
    4,3,5,1,0,2,1,0,2,9,9,9,9,9,9,9,
    5,5,4,2,2,1,2,2,1,9,9,9,9,9,9,9,
    4,3,3,1,0,0,7,6,6,9,9,9,9,9,9,9,
    4,3,5,1,0,2,7,6,8,9,9,9,9,9,9,9,
    5,5,4,2,2,1,8,8,7,9,9,9,9,9,9,9,
    7,6,6,7,6,6,4,3,3,9,9,9,9,9,9,9,
    7,6,8,7,6,8,4,3,5,9,9,9,9,9,9,9,
    8,8,7,8,8,7,5,5,4,9,9,9,9,9,9,9
};

#define UNR_CRZ(trans,sf1,sf2)W am=a%sf1,ad=a/sf1,dm=d%sf1,dd=d/sf1; \
    r+=k*trans[am+sf2*dm];a=ad;d=dd;k*=sf1;
INLINE W mcrz(W a, W d)_(W r=0,k=1;F_(SZ/2,UNR_CRZ(crz2,9,16))
    if(SZ&1){UNR_CRZ(crz,3,4)}r;)
INLINE W mrot(W x)_(W t=END/3,b=x%t,m=b%3,d=b/3;d+m*(t/3)+(x-b))
```

This implementation of the crazy operation and rotations is particularly efficient, as they are $O(1)$ operations that do not involve any expensive instructions such as `DIV` due to the modular multiplicative inverse optimisation performed by most compilers[10] (whereby division operations become multiplication operations as a result of the laws of modular arithmetic[11]). As the LUT is padded and the operation's loop is unrolled, the crazy operation becomes even faster. `UNR_CRZ` takes the crazy operation LUT and scale factors for extraction. The following `F_` loop is manually unrolled to use the larger `crz2` table, and fixed up with an `if(SZ&1){...}` clause which uses the smaller `crz` table - the operation is a fix-up on a single trit[12]. When compiled with `clang`[13], the `mcrz` function's control flow is flattened[14]. No significant performance improvements were observed when the crazy operation was unrolled further via compiler pragmas or the LUT was enlarged.

1.1.2. Memory management

A vector is the best data structure for holding the Malbolge memory. It offers $O(1)$ random access and insertion or removal of elements at the end with an amortised constant complexity $O(1)$. However, employing a vector with traditional qualities is not an efficient solution, as resizing it may cause reallocations, and allocating $4 * 3^{19}$ bytes of memory upfront (approximately 4.6 GB) is wasteful and requires filling it with a pattern derived from the source code, which negatively affects startup performance. fast20 bundled with MalbolgeLISP v1.1 solves the problems with usual vectors in the following way:

```
static unsigned int last_initialized;

static inline Word* ptr_to(Word** mem[], Word d) {
    if ((mem[d.area])[d.high]) {
        return &(((mem[d.area])[d.high])[d.low]);
    }
    (mem[d.area])[d.high] = (Word*)malloc(59049 * sizeof(Word));
    Word repitition[6];
    repitition[(last_initialized-1) % 6] =
            ((mem[0])[(last_initialized-1) / 59049])
                [(last_initialized-1) % 59049];
    repitition[(last_initialized) % 6] =
            ((mem[0])[last_initialized / 59049])
                [last_initialized % 59049];
    #pragma GCC unroll 6
    for (unsigned int i=0;i<6;i++) {
        repitition[(last_initialized+1+i) % 6] =
                crazy(repitition[(last_initialized+i) % 6],
                    repitition[(last_initialized-1+i) % 6]);
    }
    unsigned int offset = (59049*((unsigned int)d.high)) % 6;
    for(unsigned int i = 0; i < 59049; i++) {
        ((mem[d.area])[d.high])[i] = repitition[(i+offset)%6];
    }
    return &(((mem[d.area])[d.high])[d.low]);
}
```

The `ptr_to` routine is called in the following parts of the code for everything related to Malbolge memory access. It's far from ideal, since the code uses multiple pointer indirections and performs the allocation check each time a memory cell is requested (by checking just the high word; the code allocates 3^{10} worth of cells on

[10] explained in more detail on `https://kamila.akagi.moe/posts/leap-gcd/`

[11] `https://en.wikipedia.org/wiki/Modular_multiplicative_inverse`

[12] This model has been employed for the first time in `https://github.com/dzaima`'s interpreter

[13] `clang kiera-tests/fast20.c -O3 -mtune=native -march=native -fvisibility=hidden -o fast20`

[14] `https://paste.m.akagi.moe/~kamila/2382dfe7d51f0177b4abb89da385efad2e710e8c`

each fault). Essentially, the code splits the Malbolge memory into chunks, so no reallocations are required. The relevant relative cycle estimation data obtained across `callgrind` profiling sessions follows:

- 14.80 - instruction decode
- 14.64 - memory presence check + 5.92 - returning the reference (20.56 in total for `ptr_to`)
- 13.81 - incrementing/decrementing Malbolge words (mainly caused by split between high/low)
- ±5 - cumulative for crazy operation

New fast20 takes advantage of virtual memory, anonymous mapping and catching access violation exceptions to solve the problem of reallocations and checks without splitting the memory into chunks, worsening the performance because of pointer indirections. The relevant code fragment follows.

```
P u64 pgsiz;
P W*mem,pat[6];

P void mpstb(void*b,u64 l) {
  mmap(b,l,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANON|MAP_FIXED,-1,0);
}

P void sigsegvh(int n,siginfo_t*si,void*_) {
  void*a=si->si_addr,*ab=(void*)((u64)a&~(pgsiz-1));mpstb(ab, pgsiz);
  W* curr=ab;i64 off=(curr-mem)%(END/3);F1(pgsiz,sizeof(W),*curr++=pat[off++%6]);
}

P u64 rup(u64 v)_(((v-1)&~(pgsiz-1))+pgsiz)

__attribute__((hot,flatten))int main(int argc, char* argv[]){
  pgsiz=sysconf(_SC_PAGESIZE);
  mem=mmap(NULL,END*sizeof(W),PROT_NONE,MAP_NORESERVE|MAP_PRIVATE|MAP_ANON,-1,0);
  struct sigaction act;memset(&act,0,sizeof(struct sigaction));
  act.sa_flags=SA_SIGINFO;act.sa_sigaction=sigsegvh;sigaction(SIGSEGV,&act,NULL);
  FILE*f=fopen(argv[1],"rb");fseek(f,0,SEEK_END);u64 S=ftell(f);rewind(f);
  u64 szR=rup(S),off=0;mpstb(mem, szR*sizeof(W));
  /* [snip] */
  INS_4:c=*d;NXT;
  INS_5:putchar(a);fflush(stdout);NXT;
  INS_23:;int CR=getchar();a=CR==EOF?END-1:CR;NXT;
  INS_39:a=*d=mrot(*d);NXT;
  INS_40:d=mem+*d;NXT;
  INS_62:a=*d=mcrz(a, *d);
  INS_68:NXT;
  INS_81:return 0;
  INS_DEF:NXT;
}
```

The interpreter maps the entire memory area as virtual memory and registers an access violation exception handler. This way, if a write happens to the memory which isn't marked as writable (first `mmap` call sets `PROT_NONE`), the handler maps this memory as writable and then fills it with the pattern. Due to performance concerns, the code marks the entire file size as writable using `mpstb`. `mpstb` only changes the **m**emory **p**rotection, but the `mprotect` function was not employed to facilitate this, as it appears to exhibit inferior performance characteristics when called repetitively[15].

[15]The author suspects that it's caused by inter-processor interrupts causing TLB shootdowns.

It has been observed that many instances of fast20 requesting memory from the kernel via `mmap` behave poorly from within QEMU virtual machines running the latest Linux kernel available in Debian repositories (at the time of writing, 5.10.0). A single call may take up to 7 minutes and trigger the kernel timeout watchdog, displaying a soft lockup warning that declares a CPU as having been stuck for approximately 24 whole seconds. According to the `perf` utility running on the virtual machine's host, the host kernel appears to be stuck in a spinlock. These negative performance characteristics are not the fault of a programming error in fast20, and these issues are not exhibited on the host operating system.

1.1.3. Code evaluation

The old fast20 implementation's performance problems are mainly caused by an inefficient data representation, although there are improvements to be made in the evaluation function.

```
static inline unsigned char get_instruction(Word** mem[], Word c) {
    Word* instr = ptr_to(mem, c);
    unsigned int instruction = instr->low;
    instruction =
        (instruction+((unsigned int) c.low) + 59049 * ((unsigned int) c.high) +
            (c.area == 1 ? 52 : (c.area == 2 ? 10 : 0))) % 94;
    return instruction;
}

__attribute((flatten)) int main(int argc, char* argv[]) {
    Word** memory[3];
    int j;
    #pragma GCC unroll 3
    for (unsigned char i=0; i<3; i++) {
        memory[i] = (Word**)malloc(59049 * sizeof(Word*));
        for (j=0; j<59049; j++)
            (memory[i])[j] = 0;
    }
    Word a, c, d;
    FILE* file = fopen(argv[1],"rb");
    fseek(file, 0, SEEK_END);
    unsigned int size = ftell(file);
    rewind(file);
    a = zero();
    c = zero();
    d = zero();
    while(1) {
        if(__builtin_expect(size > 16, 1)) {
            /* snip: loop unrolling */
        } else {
            Word* cell = ptr_toz(memory, d);
            (*cell) = zero();
            fread(&cell->low,1,1,file);
            if (cell->low != ' ' && cell->low != '\r' && cell->low != '\n')
                d = increment(d);
            break;
        }
    }
    fclose(file);
    for(; d.low != 59048; d = increment(d)) {
        *ptr_toz(memory, d) = crazy(*ptr_toz(memory, decrement(d)),
```

```
                *ptr_toz(memory, decrement(decrement(d))));
    }
    last_initialized = 59047 + 59049*((unsigned int) d.high);
    d = zero();

    while (1) {
        unsigned char instruction = get_instruction(memory, c);
        switch (instruction){
            case 4:
                c = *ptr_to(memory,d);
                break;
            /* snip: I/O */
            case 39:
                a = (*ptr_to(memory,d)
                        = rotate_r(*ptr_to(memory,d)));
                break;
            case 40:
                d = *ptr_to(memory,d);
                break;
            case 62:
                a = (*ptr_to(memory,d)
                        = crazy(a, *ptr_to(memory,d)));
                break;
            case 81:
                return 0;
            default:
                break;
        }

        Word* mem_c = ptr_to(memory, c);
        mem_c->low = translation[mem_c->low - 33];

        c = increment(c);
        d = increment(d);
    }
}
```

The new fast20 interpreter improves upon this design by making Malbolge's `D` register an actual pointer, which offers significant improvements in performance. Representing the `C` register as a pointer is, however, not viable - as the instructions are determined based on their position within the file, so the value of `C` relative to the memory base would have to be computed upon each instruction decode. The translation is done via the padded version of the original `xlat`[16] array, as presented below:

```
W c=0,a=0,*d=mem;
P C int offs[]={
  0,
  ((i64)a1_off-(i64)(END/3))%94+94,
  ((i64)a2_off-(i64)(2*(END/3))%94+94)
};
P C void*j[94];F_(94,j[i]=&&INS_DEF)
#define M(n) j[n]=&&INS_##n;
M(4)M(5)M(23)M(39)M(40)M(62)M(68)M(81)
#define BRA {goto*j[(c+mem[c]+offs[c/(END/3)])%94];}
```

[16]Ben Olmstead's terminology; the array might have been named after a x86 instruction.

```
BRA;
#define NXT mem[c] = \
  "SOMEBODY MAKE ME FEEL ALIVE" \
  "[hj9>,5z]&gqtyfr$(we4{WP)H-Zn,[%\\3dL+Q;>U!pJS72FhO" \
  "A1CB6v^=I_0/8|jsb9m<.TVac`uY*MK'X~xDl}REokN:#?G\"i@" \
  "AND SHATTER ME"[mem[c]];c++;d++;BRA
INS_4:c=*d;NXT;
INS_5:putchar(a);fflush(stdout);NXT;
INS_23:;int CR=getchar();a=CR==EOF?END-1:CR;NXT;
INS_39:a=*d=mrot(*d);NXT;
INS_40:d=mem+*d;NXT;
INS_62:a=*d=mcrz(a, *d);
INS_68:NXT;
INS_81:return 0;
INS_DEF:NXT;
```

1.1.4. Interpreter profiling results

The profiling results for new fast20 differ from the previous results, since many aspects of fast20 have been optimised and profiling now reveals actual bottlenecks in the current Malbolge interpreter implementation. The relative cycle estimation data follows.

- 14.79 - crazy operation (in total).
- 14.87 - executing `JMP`.
- 9.91 - executing `CRZ` and `NOP`.
- 1.87 - executing `ROTR` and `MOVD`.

1.2. Special properties of Malbolge

One can consider hypothetical Malbolge code that, in the general case, may self-modify significantly, or just slightly. It may have a number of recognisable idioms within it, but this is not required. By making no assumptions about the source code[17], it is possible to ascertain some facts about it.

As polymorphic code in Malbolge is a ubiquitous phenomenon, it is not possible to perform a static disassembly of a given Malbolge program. It is also impossible to determine the instruction cycles used by the program, and because Malbolge is a von Neumann machine, it is also impossible to separate self-modifying code from static (initialised) data.

All of the issues presented here may be solved by making the assumption that the given code is executed from within a special interpreter. This hypothetical special interpreter would attempt to reconstruct a form of Malbolge assembly with a notion of initialised data, `bss` data and instruction cycle specifications. The unfortunate fact of the matter is that, to fully reconstruct this code, it is required to satisfy all of its possible code paths. Due to the effects of self-modification and lack of separation between code and data, this may not even be feasible at all.

Observation of the fast20 interpreter's code provokes an interesting thought - the Malbolge source code loaded into the Malbolge memory is a sequence of bytes (e.g. for MalbolgeLISP, around 300 MiB of data). The memory consists of double-words, thus producing 3 wasted bytes of space for each given Malbolge instruction; the problems caused by this become more visible as the size of the program increases[18]. One potential solution to this problem may be to mark certain areas of memory as code - this would reduce their size, but for more pathological cases[19], the performance would degrade significantly by the process of unpacking the code to allow for self-modification, as

[17] ... which isn't a wrong assumption to make, given how unusual Malbolge code must be to circumvent all the language's difficulties

[18] currently, because of this, MalbolgeLisp requires around 1.2GiB of memory, instead of around 400MB which would definitely be sufficient

[19] Most Malbolge programs

well as keeping data within the code area[20]. The packing process would also need to turn the Malbolge memory into a different data structure, as the requirement of each element within a vector being of equal size isn't satisfied.

One optimisation technique that has been proven to work for many programming languages is known as idiom recognition, and it has been successfully applied to languages such as Brainfuck[21], as well as real-world compilers via peephole optimisation[22] and strength reduction techniques. High-level languages where sequences of granular operations describe an operation on data that can be executed more efficiently can also benefit from this technique[23].

However, peephole optimisations cannot be guaranteed to be sound if the code is self-modifying, or if code and data are mixed. It is not exactly known to the optimisation tool what the code executes, as the given code may be anything. The optimisation tool may be forced to simply guess that something is a code instruction cycle - and even if this guess proves to be correct, the instruction cycles may be chained to perform more complex operations, which makes this entire procedure extremely non-deterministic, or even impossible to perform. In many cases the results would be unhelpful and redundant, and potentially even inefficient. Additionally, it is also not possible to utilise parallel processing to aid in the evaluation of Malbolge source code.

The following subsections demonstrate example operations that could be optimised by an interpreter that manages to employ idiom recognition.

1.2.1. Malbolge constant load idiom

Loading immediate values in Malbolge is non-trivial (except 0, which can simply be loaded by rotating `CON0` right). It requires loading a `CON1` value and performing crazy operation on it once or three times (depending on whether or not the integer in trinary notation has any ones present - i.e. it is *tricky*; only even digits in trinary representation allow a load consisting of a single crazy operation). The following load generator is proposed:

```
to_opt←{
    o←2|n←3⊥⍸⍨1⊢ω⌽s←∊⍨⍳0
    z←s n⌽y←s ~o⌽x←s 2∘×o
    ∨/o:x y z⌽⊂z
}
```

If the integer's trinary representation T is not *tricky*, it is possible to load it via the following method (using Nagoya syntax[24]):

```
{
ENTRY: ROT CON1 REV ROT
       OPR $T REV OPR
       HLT
}
```

As a result of the first line `CON1` is loaded into `A`. The second line puts T into `[D]`, putting `crazyop(CON1, T)` into both. As all `A` trits are 1, assuming the following mapping between corresponding pairs of trits: `0 1 2` is turned into `0 0 2` per the crazy operation lookup table (`0` and `2` match and `1` doesn't match, which is this method produces correct results only for numbers that aren't *tricky*), it was possible to load the immediate.

For *tricky* immediate values, a slightly different strategy is used. If there was a way to load a trinary number to A, so that it's `CON1` which has a zero trit corresponding to every one trit in the desired result[25], the same technique could be used since according to the table, `opr(0t, 1t) = 1t`.

[20] 1B→1B representation gets turned back into 1B→4B, forcing a memory move and complicating the memory map, greatly decreasing the performance of other functions that require access to the Malbolge memory, unless special cases for the access violation handler are added that make sure that the code pages are marked as read-only, and then unmarked and unpacked once a write is requested

[21] `https://github.com/rdebath/Brainfuck/tree/master/tritium`

[22] `https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gccint/Peephole-Definitions.html`

[23] `http://docs.dyalog.com/14.0/Dyalog%20APL%20Idioms.pdf`

[24] `https://www.trs.css.i.nagoya-u.ac.jp/projects/Malbolge/papers/IPSJ-SIGPRO-2014-1-6.pdf`

[25] e.g. to load `12210t`, `A = 1111101101t` to make the crazy operation presented above work

To load the mask, one should perform the crazy operation of a trinary number `CON0` with a `2` trit on every corresponding position where a `1` trit in the mask is desired. Then, to obtain the final mask value, the `1` trits from the initial number are picked and put on their corresponding positions to a `CON0`-based number, and then the number is negated[26], and the two numbers obtained are used as parameters to crazy operation. For example, to load the number `12210t`, the following sequence of operations would be required:

```
rot 1111111111t
op of 1111111111t and 20020t       # 20020t
op of 20020t and 1101t             # 1111101101t
op of 1111101101t and 12210t       # 12210t
```

Verifying the sequence with the generator ($156_{10} \Leftrightarrow 12210_3$):

```
      to_opt 156
┌─────┬─────┬─────┐
│20020│01101│12210│
└─────┴─────┴─────┘
```

1.2.2. Malbolge flag idiom

Flags are made out of an arbitrary ($N \geq 1$) amount of `NOP` instructions, followed by a `MOVD` instruction in a single cycle, and then a `JMP` instruction. Flags are set and unset by restoring and execution, and their behavior is probed by arranging the source code so that the `MOVD` instruction affects the control flow in a meaningful way. Multistate flags ($N > 2$) can be used for storing multiple values. For example, a `NOP/NOP/NOP/MOVD` chain can have the following states:

- `NOP/NOP/NOP/MOVD`
- `NOP/NOP/MOVD/NOP`
- `NOP/MOVD/NOP/NOP`
- `MOVD/NOP/NOP/NOP`

Although the multistate flag technique is rarely utilised within MalbolgeLISP, the entire codebase consists of around 14'000 `NOP/MOVD` flags.

1.3. Handling Malbolge code

Malbolge code in its unprocessed form has extremely poor compressibility characteristics. The entropy statistics of the MalbolgeLISP source code[27] as provided by the `ent` utility in the Debian Linux software repositories are as follows:

```
% ent lisp.mb
Entropy = 6.554589 bits per byte.

Optimum compression would reduce the size
of this 339823649 byte file by 18 percent.

Chi square distribution for 339823649 samples is 585653786.63, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 79.5007 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is 0.507535 (totally uncorrelated = 0.0).
```

[26] 0 trits become 1 trits, 1 trits become 0 trits

[27] Exact size: `339823649` bytes.

Despite the virtual machine only being capable of executing 6 distinct instructions, we see a demonstration of Malbolge code having an immensely high entropy. This can be explained primarily through the instruction encoding, as each instruction depends on its position within the file, and must be picked through the `xlat` lookup table (or alternatively via an inline padded string as is demonstrated by fast20, removing the need to perform a subtraction operation; in fast20 this instruction translation process occurs within the `NXT` macro). The Malbolge code is designed to mitigate this issue already, but it can also be *normalised*[28] - during this process, instructions are to be decrypted to match the original specification[29]. The following code can be used to facilitate the conversion between Malbolge and its normalised form:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

const uint8_t * xlat1 = "+b(29e*j1VMEKLyC})8&m#~W>qxdRp0wkrUo[D7,X"
    "TcA\"lI.v%{gJh4G\\-=O@5`_3i<?Z';FNQuY]szf$!BS/|t:Pn6^Ha";

uint8_t assembly(uint8_t c) {
    switch(c) {
        case 'i': return 4;
        case '<': return 5;
        case '/': return 23;
        case '*': return 39;
        case 'j': return 40;
        case 'p': return 62;
        case 'o': return 68;
        case 'v': return 81;
    }

    fprintf(stderr, "invalid character: %d (%c)\n", c, c);
}

uint8_t decodeInt(uint8_t code, uint64_t position) {
    return xlat1[(((uint64_t) code) - 33ull + position) % 94];
}

void normalize(void) {
    uint64_t t = 0; int8_t ct;
    while((ct = getchar()) != EOF) {
        if(ct > 32)
            putchar(decodeInt(ct, t));
        t++;
    }
}

uint8_t encodeInt(uint64_t code, uint64_t position) {
    int8_t t = (code - position % 94 + 94) % 94;
    if(t < 33)
        t += 94;
    return t;
}

void assemble(void) {
```

[28] https://web.archive.org/web/20170815102152/https://acooke.org/malbolge.html

[29] raw printable ASCII characters, without encoding

```
    uint64_t t = 0; int8_t ct;
    while((ct = getchar()) != EOF) {
        if(ct > 32)
            putchar(encodeInt(assembly(ct), t));
        t++;
    }
}

int main(int argc, char * argv[]) {
    if(argv[1][0] == 'e')
        assemble();
    else if(argv[1][0] == 'd')
        normalize();
}
```

It should be noted that normalised storage of Malbolge code in-memory would not benefit the interpreter unless the interpreter's memory is transparently compressed by the kernel via modules such as `zram`, as the added overhead of encode and decode operations would drastically worsen runtime performance characteristics. It is, however, apparent that this normalisation and compression process will improve the cost of transporting Malbolge programs between computers.

The following results table presents the results of MalbolgeLISP compressed in its unmodified state with various popular compression algorithms:

		Compression time		Decompression time	
Algorithm	**Compression Ratio**	**Average**	**Median**	**Average**	**Median**
PPMd mx=5	22.13958886	10.7432	10.796	13.4947	13.345
PPMd mx=9	56.35555342	23.591	23.5785	25.5343	25.426
Deflate mx=5	9.183000077	64.624	64.64	2.217	2.1185
Deflate mx=9	9.464356353	443.03	441.985	2.4009	2.396
BZip2 mx=5	27.66938453	4.0545	3.972	10.3362	10.251
BZip2 mx=9	27.70319478	19.0515	19.111	9.8867	9.761
LZMA mx=5	23.23954245	79.17	79.275	2.2699	2.1905
LZMA mx=9	28.1603411	177.241	177	2.2187	2.145
GZip -6	9.020591691	12.2658	12.2315	1.4981	1.4975
GZip -9	9.048854476	14.9222	14.8975	1.5072	1.4805

Table 1.3: Compression benchmark results

Upon normalisation of the MalbolgeLISP source code, the statistics provided by `ent` indicate much more compressible entropy figures, suggesting that the aforementioned compression algorithms could perform significantly better on normalised Malbolge code:

```
% ent lisp-norm.mb
Entropy = 1.894916 bits per byte.

Optimum compression would reduce the size
of this 339823649 byte file by 76 percent.

Chi square distribution for 339823649 samples is 24918677952.04, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 96.0930 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is -0.179165 (totally uncorrelated = 0.0).
```

In the previous compression tests, the best compression ratio figures were provided by the PPMd and LZMA algorithms at their maximum supported compression levels; consequently, only these two algorithms and levels will be tested on the normalised output:

- PPMd improved in compression time from an average of 23.6 down to 4.26 seconds, producing a 4845 KiB archive with a compression ratio of 68.5, a clear advantage over the original of 56.4.
- LZMA yielded little to no improvement in compression time from the control test, but produced a 7575 KiB archive with a massively improved compression ratio of 43.8, far above the control value of 28.2.

1.4. Arithmetic in Malbolge

Every previous attempt at implementing arithmetic in Malbolge[30] the author is aware of was bounded. The slightly modified implementation of addition used in Nagoya toolchain follows:

```
//桁上げを考慮しない加算
// x = x + y
//yは破壊される
void sum(Block* block, Variable* x, Variable* y){
  auto temp = get_temporary_variable(TYPE::INT);
  block->reset_to_con1(temp);
  block->rot(CON2)->opr(x)->opr(temp)->opr(temp)
       ->rot(CON0)->opr(x)->rot(CON2)->opr(temp)
       ->rot(CON2)->opr(y)->rot(CON2)->opr(y)
       ->opr(x)->opr(y)->opr(temp)->opr(x);
  release_temporary_variable(temp);
}

//y = (x + y)のcarry
//xは破壊されない
void carry(Block* block, Variable* x, Variable* y){
  auto temp = get_temporary_variable(TYPE::INT);
  block->reset_to_con1(temp);
  block->rot(CON2)->opr(x)->rot(CON2)->opr(x)->opr(temp)
       ->opr(y)->opr(temp)->opr(y)->rot(CON0)->opr(y)->opr(y);
  release_temporary_variable(temp);
}

void add(Block* block, Variable* x, Variable* _y){
  auto c = copy_to_temporary(block, _y);
  auto inner_block = new Block();
  auto c2 = get_temporary_variable(TYPE::INT);

  //最上位の桁上げを消すために使う変数
  auto reset = get_const_variable(TYPE::INT, 2905653667);
      // 2905653667 <=> 21111111111111111111t
  copy(inner_block, c, c2);
  carry(inner_block, x, c);
  inner_block->rot(CON2)->opr(reset)->opr(c)
             ->rot(CON2)->opr(c)->
             ->rot(CON2)->opr(reset);
  sum(inner_block, x, c2);
  inner_block->rot(x)->rot(reset);
```

[30] https://lutter.cc/malbolge/digital_root.mal

```
    block->repeat(20, inner_block);
  }
```

It is clear (judging by the last line of `add` in `parse.yy`), that this routine will not operate correctly on values larger than 20 trits, as a truncation will occur. The language is claimed to target Malbolge20[31], which explains the deliberate decision to perform arithmetic operations with a constant size, as it is faster than the alternate method involving a repetitive decrement of the source and increment of the destination.

Due to a lack of research targeted towards Malbolge Unshackled, MalbolgeLISP is the first Malbolge program to perform the addition and subtraction of arbitrary precision numbers[32], for as long as the code is executed within an interpreter that is compliant with the Malbolge Unshackled standard[33]. Since MalbolgeLISP uses a `decrement` → `increment` (or `decrement`) → `loop` implementation of arithmetic, meaning that `(+ N M)` is faster than `(+ M N)` if $N > M$[34].

MalbolgeLISP's default approach is to perform bounded (modular) arithmetic operations. This approach simplifies the management of the stack and heap - the stack grows downwards and the heap grows upwards, meaning the stack must be placed at a high memory location to fully utilise the pointer width[35].

It may be suggested that MalbolgeLISP could implement bounded addition via the Nagoya University approach, however, it negatively affects the Malbolge code size and proves difficult to integrate with the rest of the interpreter. The code would need to be adjusted to perform trinary computation[36], and the entire interpreter would need to switch from a maximum addressable memory region size of $2^{26} - 1$ to $3^N - 1$.

[31] `https://git.trs.css.i.nagoya-u.ac.jp/malbolge/highlevel`

[32] available as primitives `+'` and `-'`; the rotation width problem is solved by embedding an extension loop pass into each iteration of the operation

[33] fast20 isn't one, since it fixes the rotation width

[34] because addition's computational complexity in this case is $O(N)$ for the second argument

[35] otherwise the heap could clash into the stack or vice versa without reaching the memory at the top

[36] ensuring correct handling of overflows and uniform behavior between multiplication/division and subtraction/addition, since MalbolgeLISP doesn't implement multiplication and division using repeated addition or subtraction

Chapter 2

The Lisp interpreter

The size of the MalbolgeLISP interpreter is approximately 350 MB, and it requires a minimum of 1.5 GB of system memory to facilitate optimal usage conditions. The default arithmetic operators work on 26-bit wide integers, which is also MalbolgeLISP's pointer size; this means that the interpreter is capable of addressing around 67 million 26-bit words of memory. It is generally recommended to execute MalbolgeLISP under a fixed rotation width interpreter locked at 19 trits for the best possible performance characteristics without trading functionality.

After execution, MalbolgeLISP displays a banner similar to the one below:

```
MALBOLGELISP V1.2 (2020-2021, PALAIOLOGOS)
DOT COMMANDS: .F(eatures) .A(uthor) .R(eset) .M(emory) .S(ymbols) .E(xport) .I(mport)
%
```

The MalbolgeLISP REPL accepts two kinds of data - dot commands and MalbolgeLISP expressions. The following is the feature list reported by the interpreter, consisting of 76 builtin functions:

```
% .F
' ~ + - % < > = ! & | * / ^ def defun lambda
cond print atom cons car cdr if let defmacro
iota size nth map filter eval append append'
>= <= max min /= rev any every sort zip id
flatten flatmap tie fold zipwith iterate iterateN
where replicate count take take' drop drop' bind
atop intersperse scan monad dyad selfie commute
fold' scan' uniq off +' -' fork lazy lift bruijn
```

2.1. MalbolgeLISP's memory model and dot commands

MalbolgeLISP's memory is split between the heap and the stack. The stack is located at the end of addressable memory and grows downwards, while the heap grows upwards. The space in the heap is allocated using a bump allocator with support for rewinding. The current memory usage can be checked using the `.M` dot command, as presented below:

```
% .M
6W USED
% # some operations follow...
% .M
125W USED
```

Clearing the memory is possible using the `.R` dot command. `.R` resets the bump allocator's internal pointer to the default value, removes *all* objects and clears the global table. A few objects are preallocated on the heap[1].

[1] strings: `quote`, `lazy`, `x`, `y`; numbers: `1` - true, `0` - false

It is possible to create a saved interpreter state using the `.E` dot command, which dumps the data pointers (to preallocated data, global table pointer and interned string list), dump size and the resulting data starting from the beginning of the memory to the logical end of used memory, as indicated by the bump allocator's internal pointer. Loading the image is to be performed via the use of the `.I` dot command. The loaded data is not checked for validity, nor is it compacted before exporting or after importing. This procedure can be performed by external software, knowing the format of the memory dump.

The following data structures are employed by the MalbolgeLISP interpreter:

- An atom, which is a tagged union of all the possible data types. The mappings between type and the tag follow[2]:
  - 0 - Number
  - 1 - Unbound atom
  - 2 - List
  - 3 - Closure
  - 4 - Macro
  - 5 - Partially applied function object produced by `bind`
  - 6 - Partially applied function object produced by `bind'`[3]
  - 7 - Function composition object produced by `atop`.
  - 8 - Fork object produced by `fork`.
  - 9 - NTH object produced by the `#N` syntax.
  - 10 - Tack object produced by the `$N` syntax.
  - 11 - Lazily evaluated value.
  - 12 - Nothing (*NULL*, which is also a valid list)
  - 13+ - Invalid
- An unbound atom, which is a pointer to the first character of an interned string.
- The interned string table is a linked list of unbound atoms. It is built at parse time, such that every string atom referenced in the code is placed in the interned string table, which reduces memory usage[4].
- A list, which is a pointer to a tuple consisting of the list head, which is an atom, and the list tail, which is the next list node.
- A closure is a symbol table tied to the list containing the list of arguments and the list with code.
- A symbol table is a pointer to a triplet of the key, which is a pointer to the first character of the string representing the entry, the value, which is an atom, and the pointer to the next symbol table entry.
- A macro is a list consisting of the argument list and the code list.
- Partially applied function objects of both kinds are the beheaded lists along `bind` and `bind'`.
- Function composition objects consist of a beheaded list passed along `atop`.
- Fork objects consist of a list of functions produced by beheading the list passed along `fork`.
- nth and tack objects are numbers that can be created only during the parse time.
- Lazily evaluated values are created using the `lazy` built-in word. Lazily evaluated value is a list with code to which the caller's environment has been bound.

[2] When the `atom` function is used, it returns one of the following integers that correspond to a type

[3] `((bind f x y) z)` ⇔ `(f x y z)`, while `((bind' f x y) z)` ⇔ `(f z x y)`

[4] this is because every string at the moment of creation is compared with every existing string within the interned string table. A runtime performance cost is incurred due to this technique to reduce memory usage.

The exported memory blob starts with the following header:

- The pointer to the global table
- The pointer to the `quote` atom.
- The pointer to the `lazy` atom.
- The pointer to the `fork` atom.
- The pointer to the `x` atom.
- The pointer to the `y` atom.
- The pointer to the `1` (truthy value) atom.
- The pointer to the `0` (falsy value) atom.
- The pointer to the first interned string table entry.
- The dump size in words.

On the transport layer, the exported memory format is a sequence of five byte ASCII representations of the 26-bit values used by the interpreter, where the first four characters each map in total 24 bits, and the last character maps the remaining 2 bits.

The data is extracted from the number starting from the back. Then, they are encoded into ASCII values using the following LUT:

```
&l_n2a
txt "QWERTYUIOPASDFGHJKLZXCVBNM0123456789!@#$%^&*()_+-=[]cb;':/?.>,<a"
```

The following LUT is used for turning ASCII values minus 33 back to numbers:

```
&l_a2n
$(gen_vec({
  36,0,38,39,40,42,55,44,45,43,47,61,48,59,57,
  26,27,28,29,30,31,32,33,34,35,56,54,62,49,60,
  58,37,10,23,21,12,2,13,14,15,7,16,17,18,25,24,
  8,9,0,3,11,4,6,22,1,20,5,19,50,0,51,41,46,0,63,
  53,52}))
```

The symbols that are defined in the global table of the current session are listed using the `.S` dot command, as illustrated below:

```
% ; Binding a few variables
% (def succ (monad [x + 1]))
................|..
(lambda (x) (+ x 1))
% (def pred (monad [x - 1]))
................|..
(lambda (x) (- x 1))
% (def numid (atop succ pred))
...........|......
bind/syn
% (def x 5)
.......|..
5
% ; Checking the memory usage and
```

```
% ; printing the symbol table.
% .M
199W USED
% .S
x numid pred succ
%
```

The last command the interpreter claims to recognise is .A, shown as follows - although there exists one more undocumented dot command (besides the ones listed in the REPL's banner) that is outside the scope of this document:

```
% .A
kamila szewczyk (https://kamila.akagi.moe/), fall 2020 - fall 2021.
```

Unrecognised dot commands cause the interpreter to quit, although the preferred way of quitting the interpreter is `(off)`.

```
% .Z
Eh?
```

2.2. Parsing and evaluation overview

The MalbolgeLISP grammar is defined as follows:

```
toplevel = atom / dc
dc = "." [A-Z]
atom = _ (null / comment / number / list / quote / lazy / string / nth / tack)
null = "null"
string = [^ \n\[\]\(\)]+
nth = "#" number
tack = "$" number
quote = "'" atom
lazy = "?" atom
list = parlist / sqlist / forklist
parlist = "(" listbody
sqlist = "[" sqlistbody
forklist = "{" forklistbody
listbody = (atom listbody) / ")"
sqlistbody = (atom sqlistbody) / "]"
forklistbody = (atom forklistbody) / "}"
comment = ";" [^\n]+ "\n" atom
number = [0-9]+
_ = [ \n]?
```

`null` is a special construct. It can be used as an empty list with most primitives (like `'()`), and is generally yielded by operations like `cdr` on a single-element list. This grammar describes the input deemed acceptable by the interpreter, which means that it is able to parse dot commands as well as Lisp expressions.

Square bracket lists have different semantics from normal lists. Namely, `[a f b c ...]` ⇔ `(f a b c ...)`[5]. Conversion between square brace syntax and normal list syntax is performed at parse time and yields no additional run-time performance cost.

When an expression is entered, the interpreter begins the parsing stage. Each step of the parsing process is visually represented by a dot printed in the interpreter's output. Upon completion, a pipe is printed to separate the parsing stages from the evaluation stages, and the interpreter begins to print dots for each evaluation step.

The evaluation process is bound to the following rules in the given order:

[5] A feature borrowed from Haskell - `4 `mod` 3` ⇔ `mod 4 3`

- If the input expression proves to be a simple atom, i.e. it is *NULL*, is not a list, or the contents of the list are *NULL*:
  - If it is a bounded or unbounded string atom, then it its presence will be checked in the local value table, and then in the global value table.
  - Otherwise, the atom is returned verbatim.
  - All of these actions include special handling of lazily evaluated values.
- Otherwise, it is considered a list. Based on the type of the list's evaluated head, the following operations are to be performed:
  - Closure - create a new table made out of evaluated lambda arguments, append the closure ancestor's symbol table, then evaluate the closure's body.
  - Bind - clone the already applied argument list, append the current argument list and evaluate the expression. Because of this design, partial application is stackable.
  - Tack - out of the list containing the tack, pick n-th element and evaluate it.
  - Unbound atom - perform a built-in operation[6].
  - Reverse bind - clone the current argument list and insert it straight after the partially applied function. Then, evaluate the expression.
  - Macro - create a new table made out of macro arguments (unevaluated) and evaluate the macro's body.
  - Fork - create a resulting list consisting of just the first function passed to fork, and for each supplied function to the fork except the first one, cons it to the argument list and append it to the resulting list. Then, evaluate the resulting list.
  - Nth - out of the list supplied as the only argument, pick the n-th element and evaluate it.
- Lazy values are evaluated when needed by a separate function, which evaluates them as lambda expressions.

It is not possible to evaluate a list with a non-callable head from within MalbolgeLISP. Consequently, `(uniq (2 3 3))` and any other expressions that make use of unquoted data lists will result in an error. Quoting (via `'(...)` or `(quote ...)`) prevents evaluation, while `eval` forces it.

2.2.1. Strict definition of equality

Equality checks within MalbolgeLISP are obedient to the following rules:

- If pointers to the atoms are equal, the atoms are compared equal.
- If both atoms are *NULL*, the atoms are compared equal.
- If the types of both atoms differ, the atoms are compared unequal.
- If any of the atoms is a number, their numerical value is compared.
- If any of the atoms is unbounded (i.e. a string), they are compared character by character.
- If any of the atoms is a list, the list is traversed and the the equality rules are applied on every list element. Only of all list elements compare equal they are compared equal.
- Otherwise, the atoms are compared unequal. For this reason[7] closures, partially applied functions, forks and atops compare unequal.

Inequality is checked by inverting the result of equality checking. There exists total ordering on numbers, but nothing else[8].

[6] like `car`, `scan` or `+`

[7] only in non-trivial cases, i.e. not outlined above

[8] hence the `sort` function works only with one-deep, numeric lists

2.2.2. Efficient built-in function recognition

MalbolgeLISP v1.2 employs a novel way of built-in function recognition, which uses hashing to accomplish it's task. Since unbound atoms can be compared and printed, not every string atom can be hashed[9]. The string atoms are hashed before mathematical operators and other single-character functions are processed[10]. The code follows:

```
;       f←3⊥97-⍨⎕ucs
;       f 'iterateN' ⍝ example usage
; 33860

; r1 => the output
; r2 => pointer to the function name
; r3 => trashed
@hash
    clr r1
@hash_loop
    lods r3, r2
    ceq r3, 0
    cret
    mul r1, 3
    sub r3, .a
    add r1, r3
    jmp %hash_loop
```

This hashing algorithm has a somewhat subtle flaw - it's prone to collisions and MalbolgeLISP was unlucky enough to be affected by it:

```
      'any' ≡⍤f 'gcd'
1
```

For this reason, the code employs an additional check for the first letter of the string atom if it's hash is equal to 63.

2.2.3. List cloning

Since list contents in MalbolgeLISP can't be mutated[11], they don't have to be cloned often. Consider the following list:

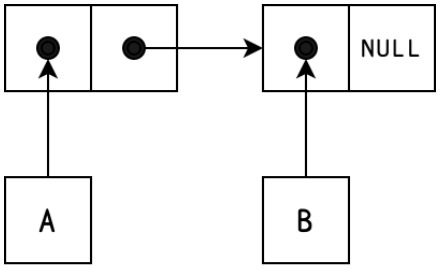

Figure 2.1: A flat two-element list consisting of unbound atoms A and B

The end of the list is marked with *NULL*, to signify that no further nodes are a part of it. Some operations on lists don't require cloning them - for instance, beheading the list (omitting the first element) can be accomplished as taking the tail of the first node in the list - all pointers to the nodes of this list are still valid and point to the desired data:

[9] doing so would carry losing entropy as a consequence

[10] since i.a. the `gcd` and `lcm` functions are hashed, even though `+` or `-` don't require hashing

[11] operations like `map` create a new list each time

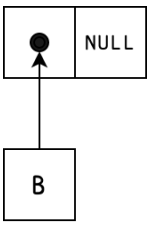

Figure 2.2: A beheaded list with only atom B left

Prepending content to a list (creating a new list node and setting it's tail) can also be done without a copy - `(cons C 1)` yields the following result:

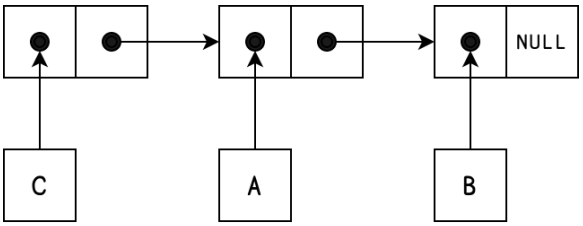

Figure 2.3: A list with atom C prepended

All pointers to the existing data are still valid and point to the same list as before. Appending to a list requires copying it, since the *NULL* tail is being replaced to the pointer with the appended node, meaning that every pointer to the list points to the modified list now, which is undesirable.

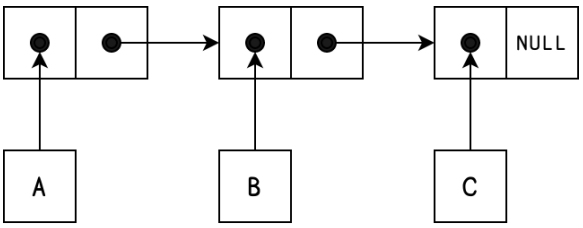

Figure 2.4: A list with atom C appended

List cloning in this case isn't deep. Considering the following deep list:

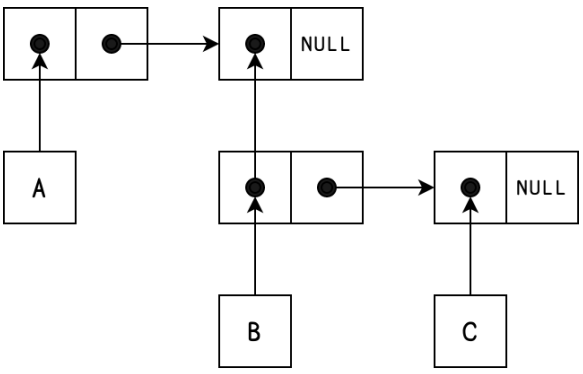

Figure 2.5: A list consisting of atom A and a list containing atoms B and C

Appending to this list involves changing the node at the end of it. As the atoms inside of it are unaffected (including nested lists), a deep clone isn't performed, since it isn't needed[12]:

[12]MalbolgeLISP doesn't perform deep clones under any circumstances

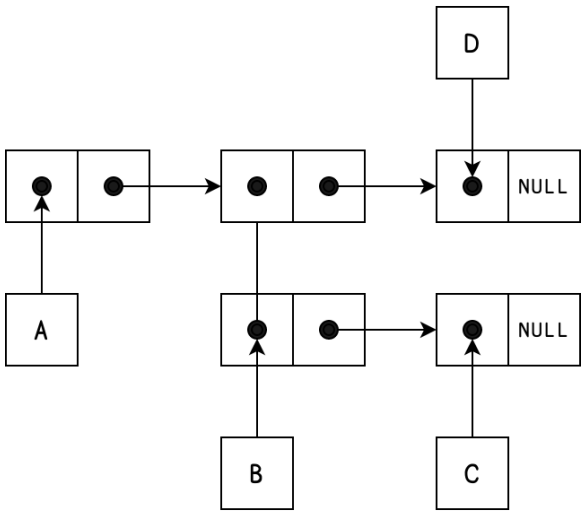

Figure 2.6: A list consisting of atom A and a list containing atoms B and C

Consequently, `cons` will always be faster than `append` or `append'` (for merging lists, `append'` will always be faster than a loop invoking `append` repeatedly, since it doesn't perform nearly as many linked list pointer indirections - $O(m+n)$ compared to $O(m(m+1)/2+n) \Leftrightarrow O(m^2+n)$).

An interesting special case of list cloning is related to the higher order functions related to point-free programming. Namely, when evaluated, `selfie`, `commute`, `lazy`, `fork`, partially applied functions, and reverse partial applications will clone their code lists. This happens since the code lists are sometimes evaluated, changed and evaluated again many times (`map`'s behavior), and the data perceived by lazily evaluated values might change in an undesirable way - for instance, `((atop #1 map) (bind lazy ~) '(0 1 1 1 1 0))` could yield `1`, while `0` was expected.

2.3. The error table

The following table lists all of the possible error messages that may be displayed by MalbolgeLISP during the evaluation of a given LISP expression:

Error code	Description
`E000`	Invalid amount of arguments passed to `if`.
`E001`	Invalid amount of arguments passed to `quote` (usually used as `'`).
`E002`	Invalid amount of arguments passed to `def`, or the first argument isn't a string atom.
`E003`	Invalid amount of arguments passed to `lambda`, or the first argument isn't a list.
`E004`	Invalid amount of arguments passed to `defun`, or the first argument isn't a string atom, or the second argument isn't a list.
`E005`	Invalid amount of arguments passed to `defmacro`, or the first argument isn't a string atom, or the second argument isn't a list.
`E006`	`bind` and `bind'` require at least two additional arguments - the function to bind to, and at least one argument to apply.
`E007`	`atop` requires at least two additional arguments - at least two functions are required for composition.
`E008`	`monad` and `dyad` require only one additional argument - the code block in which `x` or `x` and `y` should be bound.
`E009`	Invalid amount of arguments passed to an user-defined closure or function.
`E010`	The arithmetic operations require exactly two additional arguments.
`E011`	The arithmetic operations can work only on numbers, but an atom of different type was supplied.

`E012`	Invalid amount of arguments passed to `=` - only two atoms can be compared. Use `uniq` and `size` to check the equality of more atoms at a time.
`E013`	Invalid amount of arguments passed to `/=` - only two atoms can be compared.
`E014`	Invalid amount of arguments passed to `~`.
`E015`	The only argument passed to `~` isn't numeric.
`E016`	Invalid amount of arguments passed to `!`.
`E017`	The only argument passed to `!` isn't numeric.
`E018`	Invalid amount of arguments passed to `+'` or `-'`.
`E019`	The arguments passed to `+'` or `-'` aren't numeric.
`E020`	Invalid invocation of `car` or `cdr` (expected a single extra list argument)
`E021`	Invalid amount of arguments passed to `cons`.
`E022`	Invalid right argument passed to `cons` (expected a list, pairs are unsupported).
`E023`	Invalid amount of arguments passed to `atom`.
`E024`	Invalid amount of arguments passed to `print`.
`E025`	Invalid type of argument passed to `cond` (expects lists).
`E026`	Non-exhaustive `cond`.
`E027`	Invalid invocation of `let` (expected two lists).
`E028`	No `let` bindings or an unbalanced binding exists.
`E029`	Invalid amount of arguments passed to `iota`.
`E030`	Invalid type of argument passed to `iota` (expected number).
`E031`	Invalid amount of arguments passed to `size`.
`E032`	The only argument passed to `size` isn't a list.
`E034`	Invalid amount of arguments passed to `nth`.
`E035`	Invalid argument types passed to `nth` (expected a number and a list).
`E036`	Out of bounds access using `nth`, `#N` or `$N`.
`E037`	Invalid amount of arguments passed to `flatmap` or `map`.
`E038`	Invalid type of 2nd argument passed to `flatmap` or `map` (expected a list).
`E039`	Invalid amount of arguments passed to `filter`.
`E040`	Invalid type of 2nd argument passed to `filter` (expected a list).
`E041`	`filter`'s functor returned an invalid type.
`E042`	Invalid amount of arguments passed to `eval`.
`E043`	Invalid amount of arguments passed to `append`.
`E044`	Invalid amount of arguments passed to `append'`.
`E045`	Invalid type of arguments passed to `append'` (expected a non-null list as the first argument and a list as the second argument).
`E046`	Invalid type of argument passed to the monadic overload of `rev`.
`E047`	Invalid type of arguments passed to the dyadic overload of `rev` (expected a number N and a list L, where `[(size L) > N]`).
`E048`	Unrecognised overload of `rev`.
`E049`	Invalid amount of arguments passed to `any` or `every`.
`E050`	The second argument passed to `any` or `every` isn't a list.
`E051`	`any`'s or `every`'s functor returned an invalid type.
`E052`	Invalid type of argument passed to the monadic overload of `sort`.
`E053`	Invalid type of element in a list passed to the monadic overload of `sort` (monadic `sort` can sort only numbers, due to lack of total ordering on other atoms).
`E054`	Invalid type of 2nd argument passed to the dyadic overload of `sort`.
`E055`	`sort`'s functor returned an invalid type.
`E056`	Unrecognised overload of `sort`.
`E057`	Invalid amount of arguments passed to `zip`.
`E058`	Invalid type of arguments passed to `zip` (expected two lists).

`E059`	Invalid amount of arguments passed to `id`.
`E060`	Invalid amount of arguments passed to `flatten`.
`E061`	Invalid type of argument passed to `flatten` (expected a list).
`E062`	Invalid amount of arguments passed to `tie` (expected at least two arguments; to create a single element list use `cons`).
`E063`	Invalid amount of arguments passed to `where`.
`E064`	Invalid type of argument passed to `where` (expected a list).
`E065`	Invalid type of element in a list passed to `where` (expected a number).
`E066`	Invalid amount of arguments passed to `iterateN` (expected at least three arguments).
`E067`	Invalid type of amount of iterations specified in an invocation to `iterateN` (expected a number).
`E068`	Invalid amount of arguments passed to `iterate` (expected at least three arguments).
`E069`	`iterate`'s functor returned an invalid type (expected boolean).
`E070`	Invalid amount of arguments passed to `fold` or `fold'` (expected two arguments for `fold'`, or three for `fold` - the identity element).
`E071`	Invalid type of the last argument passed to `fold` or `fold'` (expected a list).
`E072`	Invalid amount of arguments passed to `zipwith` (expected three arguments).
`E073`	Last two arguments passed to `zipwith` aren't lists.
`E074`	Invalid amount of arguments passed to `replicate` (expected two arguments).
`E075`	The dyadic `list list` overload to `replicate` expects only numbers in the first list.
`E076`, `E077`	Invalid argument types passed to `replicate` (expected `list list`, `number list`, or `number number`).
`E078`	Invalid amount of arguments passed to `count` (expected two arguments).
`E079`	The second argument passed to `count` was expected to be a list.
`E080`	`count`'s functor returned an invalid type.
`E081`	Invalid amount of arguments passed to `take` or `drop'` (expected two arguments).
`E082`	Invalid type of arguments passed to `take` or `drop'` (expected `number list`).
`E083`	Invalid amount of arguments passed to `drop` or `take'` (expected two arguments).
`E084`	Invalid type of arguments passed to `drop` or `take'` (expected `number list`).
`E085`	Invalid amount of arguments passed to `intersperse` (expected two arguments).
`E086`	The second argument passed to `intersperse` was expected to be a list.
`E087`	Invalid amount of arguments passed to `scan` or `scan'` (expected two arguments for `scan'`, or three for `scan` - the identity element).
`E088`	Invalid type of the last argument passed to `scan` or `scan'` (expected a list).
`E089`	Invalid amount of arguments passed to `selfie` (expected two arguments).
`E090`	Invalid amount of arguments passed to `commute` (expected three arguments).
`E091`	Invalid amount of arguments passed to `uniq` (expected a single argument).
`E092`	Invalid argument type passed to `uniq` (expected a list).
`E093`	Unrecognised built-in function.
`E094`	Invalid user-defined macro invocation (invalid amount of arguments supplied).
`E095`	Can't evaluate (attempted to evaluate a numeric list without quoting?)
`E096`	Invalid amount of arguments passed to `fork` (expected at least two arguments).
`E097`	Invalid amount of arguments passed to constant n-th (expected a single argument).
`E098`	Invalid type of argument passed to constant n-th (expected a list).

`E099`	Invalid amount of arguments passed to `lazy` (expected at least two arguments).
`E100`	Invalid amount of arguments passed to `lift` (expected two arguments).
`E101`	Invalid type of second argument passed to `lift` (expected a list).
`E102`	Invalid amount of arguments passed to `bruijn` (expected a single argument).
`E103`	Invalid type of the argument passed to `bruijn` (expected a number).
`E104`	Not enough lambda expressions in scope for `bruijn` invocation.

2.4. Value types

MalbolgeLISP supports many built-in value types. The logical types (as recognised by the built-in functions) differ from the types actually implemented in the interpreter.

The boolean type in MalbolgeLISP doesn't exist. Instead, the numerical value `0` is considered falsy, and every other value is considered truthy. Built-in functions will always return either `0` or `1` to represent a boolean value.

The built-in higher order functions don't have the notion of a callable type. For example, `+` (outside of being an unbound atom which happens to be recognised as a built-in function) is a valid callable type, just like the result of binding, atops, forks, macros, lazily evaluated values and lambda expressions. Since they all are considered (internally) callable types, but most of them don't have a canonical representation (like lambda expressions do), they are considered *synthetic* types. When printed, the interpreter will use `bind/syn`, `macro/syn` or `lazy/syn` to represent them.

There isn't a separate type for quoted values. Instead, quoting simply prevents evaluation (since the `'(...)` syntactic sugar is translated into `(quote (...))`, and `quote` is a built-in function that returns it's argument without evaluating it).

MalbolgeLISP doesn't support pairs. For this reason, `cons` always assumes its right argument is a list (or *null*). `tie`[13] is a more convenient version of `cons`, since `(tie a b c ... z)` is always equivalent to a ladder of invocations to `cons` - `(cons a (cons b (cons c ... (cons z null))))`. Lists can also be untied (a callable object is evaluated so that an arbitrary list is treated as the parameter list). The following function accomplishes this using continued partial application:

```
(defun untie (f args) (
    (if (= null args)
        (f)
        (untie (bind f (car args)) (cdr args))))
```

A more efficient version of this function could be made using `iterate` or `iterateN`, but instead of using a Lisp implementation of `untie`, the built-in `lift` function can be used (accomplishing the same result more efficiently).

2.5. Lambda expressions, functions and macros

In MalbolgeLISP code, lambda expressions are ubiquitous as a form of binding variables (`let` uses a lambda expression in it's implementation), defining recursive functions (since point-free programming is generally considered to not be able to represent recursion, but it might be possible for some special, simple cases using `iterate` and `iterateN`[14]), or functions too complex to be represented entirely in point-free style. Sometimes they are also preferred to point-free expressions, since programmers find them harder to read[15].

Lambda expressions in MalbolgeLISP v1.2 follow the rules of lexical (static) scoping, meaning that the lexical ancestor's bound variables are visible inside the lambda expression, but they also can be shadowed using `let` bindings or `lambda` arguments. MalbolgeLISP v1.0 didn't have the concept of scoping, and a few public pre-release versions of MalbolgeLISP v1.1 employed dynamic scoping, meaning that the caller's bound variables are visible inside the lambda expression.

[13] In other implementations also called `list` - `https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node149.html`

[14] `https://dfns.dyalog.com/n_tacit.htm`

[15] `https://spin.atomicobject.com/2017/09/29/point-free-notation/`

The `defun` built-in function is defined as syntactic sugar over `def` and `lambda`, so that `(defun a (x y z ...) (...)` is equivalent to `(def a (lambda (x y z ...) (...)))`. Functions can also be defined without the lambda expression overhead[16] as tacit expressions. Monadic and dyadic functions can be defined using `monad` and `dyad` built-in functions, which are syntactic sugar over `(lambda (x) (...))` and `(lambda (x y) (...))` respectively.

In MalbolgeLISP, a macro is considered to be equivalent to a function, except it doesn't have the concept of scoping, and macro arguments aren't evaluated[17]. Since the main inefficiencies of lambda expressions don't apply to macros, they tend to be much more efficient. Macro representation also more efficient, since macros are stored as a single list of arguments and the code, without the auxiliary structure and pointer indirection to the code and cloned symbol table (as observed in lambda expressions).

2.6. Point-free programming

Point-free programming (also called tacit programming) is one of MalbolgeLISP's strong sides. It's facilitated using a wide range of advanced primitive functions[18]. Tacit programming is all about transformations on functions and their data, so that the computation can be expressed without explicitly naming the parameters and creating direct functions. MalbolgeLISP implements or faciliates the following functions and concepts:

- The monadic identity function `id`, for which `id x` $\Leftrightarrow$ `x` holds. Equivalent to `$0`.
- A partial application higher order function `bind`, for which `(bind f x y...) a b...` $\Leftrightarrow$ `f x y... a b...` holds.
- A reverse partial application higher order function `bind'`, for which the identity `(bind' f x y...) a b...` $\Leftrightarrow$ `f a b... x y...` holds.
- The variadic identity function (a *tack*[19]) - a μ-recursive primitive projection function $P_i^k(x_1, \ldots, x_k) \stackrel{\text{def}}{=} x_i$ with implied k.
- The μ-recursive primitive constant function `bind $0 n`, defined as $C_n^k(x_1, \ldots, x_k) \stackrel{\text{def}}{=} n$.
- Variadic monad lifting function `fork` (the μ-recursive substitution operator). Given an m-ary function $h(x_1, \ldots, x_m)$ - *reductor*, and m k-ary functions $g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)$ - *reductees*, it is known that $h \circ (g_1, \ldots, g_m) \stackrel{\text{def}}{=} f$ where $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$ holds.
- A variation of primitive μ-recursive recursion operator `iterateN`[20]. Given the k-ary function $g(x_1, \ldots, x_k)$ the iteration count n and k parameters r_1 up to r_k:

$$\rho(g, n, r_1, ..., r_k) \stackrel{\text{def}}{=} \begin{cases} g(\rho(g, n-1, r_1, \ldots, r_k), r_2, \ldots, x_k), & \text{for } n > 1 \wedge k \geq 2 \\ g(\rho(g, n-1, r_1)), & \text{for } n > 1 \wedge k = 1 \\ g(r_1, \ldots, r_k), & \text{for } n = 1 \wedge k \geq 2 \\ g(r_1), & \text{for } n = 1 \wedge k = 1 \\ r_1, & \text{for } n = 0 \end{cases}$$

- A variation of the `iterateN` higher order function, which decides if the function will be iterated further based on the previous and current result. For instance, `bind iterate =` is a limit operator (applying a function until the result is stable).
- A varaidic function composition operator `atop`, for which `(atop f g h...) x y...` $\Leftrightarrow$ `f (g (h... x y))` holds.

[16] caused *inter alia* by lexical scoping

[17] meaning that variables aren't expanded, lists don't need quoting, etc...

[18] tacks, constant `n-th`, `tie`, `atop`s, monad lifting - `fork`s, `bind`s and reverse binds

[19] `https://aplwiki.com/wiki/Identity`

[20] `http://help.dyalog.com/15.0/Content/Language/Primitive%20Operators/Power%20Operator.htm`

- `tie`, `lift`, and constant `n-th` functions for handling variadicity using lists.
- A concept derived from De Bruijn indices, `bruijn N` yielding the n-th lambda expression in scope, assuming 0 is the current one, 1 is it's lexical ancestor, etc...

One can build further on the mentioned concepts to introduce point-free programming idioms:

- Due to the variadic nature of `fork`, if it's invocation involves multiple functions, the resulting list sometimes can't be applied to many functions. For this reason, the idiom consisting of `tie` and `atop` is used to turn all the results into a list, to then use it with a function (which is often a `fold`) - `fork (atop f tie) g h`.
- In `fork`s, *tacks* are often used as reductees to copy a specified argument to the reductor. For instance, the following APL-style tacit inner product function utilises this idiom:

```
; MalbolgeLISP
% (def . (fork fold' $0 (fork zipwith $1 $2 $3)))
...................|.......................
(fold' $0 (zipwith $1 $2 $3))
% [+ . * '(1 2 3) '(4 5 6)]
...................|.....................................
32

⍝ APL
      1 2 3 +.× 4 5 6
32
```

- In tacit expressions, square bracket lists (denoted by `[]`) are often used, since they allow separation of `fork`'s reductor from it's reductees, improving the readability (as an alternative to `{}` syntax). They are also used alongside `bind` and `bind'`. For example, a defined average function could be translated in the following way:

```
; Average of list x is the sum of it divided by it's size. Assumes size >= 1.
(def avg (monad [[+ fold' x] / (size x)]))
(def avg (monad ([/ fork [fold' bind +] size] x)))
(def avg [/ fork [fold' bind +] size])
; Using the dedicated fork syntax.
(def avg {/ [fold' bind +] size})
```

- Tack factoring is used to optimise certain fork expressions - `(fork f (atop g $0) (atop h $0))` ⇔ `(atop (fork`
  Tack factoring also works for tack reductees, since `$0` is equivalent to `(atop id $0)`.
- Commuting sometimes allows to avoid forks - `{f $1 $0}` ⇔ `(bind commute f)`. Although, since most tacit expressions tend to involve `bind`, `bind'` can be used instead of `commute`: `{+ (bind' / 2) 32}` is more efficient than `{+ (bind commute / 2) 32}`.
- Duplicating arguments - `{f $0 $0}` ⇔ `(bind selfie f)`.
- Lifting lists - `(lift f '(x y z...))`[21] ⇔ `f x y z...`.

`commute` and `selfie` can be re-implemented in Lisp as follows:

```
(def selfie' (dyad (x y y)))
(defun commute' (x y z) (x z y))
```

Since MalbolgeLISP offers first class functions, lazy evaluation, short-circuiting inside tacit expression, arbitrary forks, atops and arbitrary tacks, MalbolgeLISP's tacit programming model is superior to APL's[22], making many concepts much easier to express.

[21] the list doesn't have to be specified inline

[22] `https://dfns.dyalog.com/n_tacit.htm`

2.7. Numerical algorithms

MalbolgeLISP implements a set of basic numerical algorithms - greatest common divisor, least common multiple, power and factorial.

The factorial function's value grows very quickly, so it can be pre-computed using a lookup table, meaning that factorials can be yielded without expensive computation. The following APL program has been used to generate the lookup table:

```
{⍵'{'('⍵,'(1↓⍳⍵,,⍕0)⍎¨m/⍨⍵≥m←!⍳20)'}'}2*26
```

The MalbolgeLISP code guards against out of bounds accesses by returning 0 on overflow:

```
&f_table
$(gen_vec({1,2,6,24,120,720,5040,40320,362880,3628800,39916800}))

@fact
    cle r2, 11
    clr r1
    cmov r1, *f_table
    cadd r1, r2
    crcl r1, r1
    ret
```

The power function implementation utilises the binary exponentiation algorithm to achieve better performance. The recursive implementation used in past versions of MalbolgeLISP follows:

```
@pow
    ceq r3, 0
    cmov r1, 1
    cret
    push r3
    asr r3
#call("pow")
    pop r3
    ; XXX: for brainfuck target, use a diff. encoding
    mul r1, r1
    mod r3, 2
    cne r3, 0
    cmul r1, r2
    ret
```

MalbolgeLISP v1.2 employs the improved, iterative version of the algorithm:

```
@pow
    mov r1, 1
@pow_loop
    ceq r3, 0
    cret
    mov r4, r3
    mod r4, 2
    ceq r4, 1
    cmul r1, r2
    ; XXX: use diff. encoding for bf target
    mov r4, r2
    mul r2, r4
    asr r3
    jmp %pow_loop
```

Greatest common divisor is implemented using an algorithm utilising two Euclidean division operations[23] and the least common multiple function is defined using the following formula:

$$\operatorname{lcm}(a,b) = \frac{ab}{\gcd(a,b)}$$

```
@gcd
    ceq r2, 0
    cret
    mod r1, r2
    cne r1, 0
    cmod r2, r1
    cjnz %gcd
    mov r1, r2
    ret

@lcm
    mov r3, r1
    mul r3, r2
#call("gcd")
    div r3, r1
    mov r1, r3
    ret
```

2.8. Laziness and side effects

MalbolgeLISP provides first-class support for lazy evaluation. Lazily evaluated expressions are introduced to the code using the `lazy` built-in word or, alternatively, the `?` list prefix. Laziness is implemented similarly to how Java[24] or C++[25] achieve it. Speaking from a high-level point of view, the computation is wrapped in a closure, and when the value is requested, the closure is evaluated and the result takes place of the lazily evaluated atom[26]. The concepts of memoization and lazy evaluation are illustrated on the following example:

```
; Define a product function that has a side effect of printing,
; so that it's known when the value is evaluated and memoized.
% (defun yelling_prod (x y) (lazy print [x * y]))
.....................|.
(lambda (x y) (lazy print (* x y)))
; Define an eager product function.
% (defun eager_prod (x y) (print [x * y]))
.....................|.
(lambda (x y) (print (* x y)))
; The lazy version took less steps to evaluate, since the
; product wasn't computed. the lazy version also didn't
; print anything, unlike the eager version.
% (def l1 (yelling_prod 9 9))
...........|.......
lazy/syn
% (def e1 (eager_prod 9 9))
```

[23] which proves itself to be faster than the standard Euclidean algorithm, due to the Malbolge toolchain intricacies

[24] via the `Supplier<T>` (`https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html`) class and wrapping the computation in a lambda expression

[25] via suspected function; explained in more detail in Appendix B - although some more sophisticated attempts were made: `https://github.com/MarcDirven/cpp-lazy`.

[26] this technique is called *memoization*

```
..........|..........
81

81
; Since the value of the lazy version is required to produce
; a result, it's evaluated (because 81 is printed).
; The evaluation was deferred.
% (print [e1 + [l1 + ?(+ 2 2)]])
..................|..............
81
......
166

166
; Because the value of l1 was memoized, the evaluation of l1
; is instant.
% e1
.|.
81
% l1
.|.
81
```

Lazy evaluation can reduce the run-time of more complex algorithms[27] - if `lazy` is omitted in the expression `((atop #1 map) (bind lazy selfie *) (iota 9))`, it takes 5 more seconds to evaluate (since the `map` functor is being applied eagerly over each of the indices, and conclusively, only the second element from the list is desired). The difference would be even more noticeable if the `map` functor was an expensive operation.

Since lazy evaluation makes the side effect handling difficult at times, an equivalent of the Haskell's IO monad could be beneficial to MalbolgeLISP, although the only side effect MalbolgeLISP supports is evaluated output (via `print`). Input isn't supported, because it's mutually exclusive with existence of the evaluation progress bar. For this reason, no way to sequence operations comparable to Haskell's IO monad or `do` blocks is supported in MalbolgeLISP.

2.9. Missing features

MalbolgeLISP is missing a few features that are generally considered convenient[28]. The following list provides a rationale behind every missing feature:

- **Negative numbers** - they would require special handling in multiplication and division routines, special handling in integer printing routines and comparison routines, causing a huge performance overhead for little to no gain.
- **String constants** - MalbolgeLISP has no *actual* strings, only unbounded atoms. Because the author's vision of string handling in MalbolgeLISP would involve orthogonal operations for ordinary lists and strings (*lists of characters*), string syntax would be implemented as syntactic sugar over lists.
- **Floating point arithmetic**, **fixed point arithmetic**, **fractional arithmetic** - floating point arithmetic would make the codebase too complex (as in the signed integer case) for very little gain, since floating point arithmetic could be replaced by the more efficient fixed point arithmetic or fractional arithmetic. Fixed point arithmetic wasn't implemented in primitives, since it would lead to breaking the orthogonality of arithmetic functions. Fractional arithmetic wasn't implemented, since it can be easily introduced to MalbolgeLISP as a set of functions, without taxing the interpreter's size.

[27] assuming one ignores the overhead of lazily evaluated values, which can sometimes make the difference too subtle or worsen the matters

[28] sometimes even essential, although they can be synthesised in MalbolgeLISP using macros

- **Pairs** - the support for them would complicate the interpreter code for no reason. They can be implemented as macros or functions with little overhead, since representing pairs as lists isn't remarkably inefficient.
- **Garbage collection or reference counting** - a lot of the MalbolgeLISP codebase depends on the properties of a bump allocator[29]. Precise reference counting in MalbolgeLISP would cause too much overhead and would be difficult to implement because of the limitations in tooling. Garbage collection suffers from the same issues, but much more exaggerated - a garbage collector would require scanning the stack and walking the entire memory, then compacting it, adjusting all the pointers and tweaking the internal bump allocator pointer. Since this garbage collector design is conservative, it's not guaranteed to be precise. Different garbage collection algorithms would require a more intricate design and resources.

[29] freshly allocated memory is zeroed, references to old objects persist, so they may be reused at some point

Chapter 3

The Language

The MalbolgeLISP implements a Lisp dialect which is described in this chapter. The main differences from Lisp is emphasis on functional programming, immutability and minimisation of the state (MalbolgeLISP doesn't implement `setcar!` or `setcdr!`). MalbolgeLISP borrows it's core features from Scheme, Haskell and APL, while using Lisp as a foundation or framework for constructing a language.

3.1. Arithmetic

MalbolgeLISP supports the following arithmetic operations:

- `+`, `+'` - unsigned bounded addition[1] and unbounded addition[2]
- `-`. `-'` - unsigned bounded subtraction and unbounded subtraction.
- `*`, `/`, `%` - unsigned bounded multiplication, division and modulus.
- `~` - boolean negation - $f(x) = 0$ for $x > 0$ and $f(x) = 1$ otherwise.
- `!` - factorial function - $f(x) = x!$ computed using a lookup table for $x \leq 11$ and $f(x) = 0$ otherwise, since !12 exceeds the word size.
- `^` - n-th power function assuming $0^0 = 1$.
- `&` - logical AND function - $f(x, y) = 1$ for $x > 0 \wedge y > 0$, $f(x, y) = 0$ otherwise.
- `|` - logical AND function - $f(x, y) = 0$ for $x = 0 \wedge y = 0$, $f(x, y) = 1$ otherwise.
- `=`, `/=` - equality and inequality.
- `<`, `>`, `<=`, `>=` - correspondingly: lesser than, greater than, lesser or equal to, greater or equal to.
- `gcd`, `lcm` - greatest common divisor and least common multiple.
- `min`, `max` - pick the smaller or larger value (*minimum* or *maximum*).

Demonstration of the arithmetic functions follows:

```
% (+ 2 1)
..............|.....
3
% (* 3 3)
..............|.....
9
```

[1] obeying to the laws of modular arithmetic; the result is truncated to 26 bits
[2] on standards-compliant interpreters

```
% (% 5 2)
..............|.....
1
% (> 5 6)
..............|.....
0
% (< 5 6)
..............|.....
1
% (= 6 6)
..............|......
1
% (= (~ (= 6 7)) (/= 6 7))
......................|.........
1
% (/ 25 5)
..............|.....
5
% (- 4 3)
..............|.....
1
% (& (= 2 2) (= 3 3))
.....................................|..............
1
% (| (= 2 3) (= 3 2))
................|..........
0
% (lcm 14 29)
......|....
406
% (max 5 (max 8 10))
...........|.......
10
% (^ 2 8)
......|....
256
% (! 6)
.....|...
720
```

3.2. Conditional execution

In MalbolgeLISP, conditional evaluation is accomplished using `if` and `cond`[3]. `&` and `|` don't short-circuit, meaning that their operands are always evaluated. `if` is very similar in it's structure to `if..else` statements[4] known from languages like C++ or Java - `(if condition valueIfTrue valueIfFalse)`. The short-circuiting aspect of `if` can be demonstrated - If both cases of the `if` condition were evaluated, the interpreter would print `yes` and `no` before yielding the correct result:

```
% (def noisy_eq (dyad (if (= x y) (print yes) (print no))))
.............................|..
```

[3] also `iterate`, `iterateN`, etc..., but these functions weren't designed for conditional evaluation and it only happened so that they can be used for this purpose, so they won't be covered in this section.

[4] or ternary operator expressions

```
(lambda (x y) (if (= x y) (print yes) (print no)))
% (noisy_eq 5 6)
......|............
no

no
% (noisy_eq 6 6)
......|............
yes

yes
```

The `cond` function is essentially a chain of `if` expressions. To demonstrate the convenience of using `cond` over multiple `if` expressions, two implementations of three-way comparison are given:

```
; An implementation that uses if.
% (def <=> (dyad (
    if (= x y) eq (
        if (> x y) gt (
            if (< x y) lt unreachable)))))
.........................................|..
(lambda (x y) (if (= x y) eq (if (> x y) gt (if (< x y) lt unreachable))))
% (<=> 5 5)
......|..........
eq
% (<=> 6 7)
......|...................
lt
% (<=> 7 6)
......|...............
gt

; An implementation that uses cond.
% (def <=> (dyad (cond
    ((= x y) eq)
    ((> x y) gt)
    ((< x y) lt)
    (unreachable))))
.............................................|..
(lambda (x y) (cond ((= x y) eq) ((> x y) gt) ((< x y) lt) (unreachable)))
% (<=> 6 6)
......|...........
eq
% (<=> 6 7)
......|...................
lt
% (<=> 7 6)
......|...............
```

It's worth mentioning that an `if` expression must include a truthy and falsy clause (since the expression must evaluate to something and it's impossible to guarantee that the condition is always true). This isn't the case with the `cond` expression. If one is certain that a `cond` expression is already exhaustive, it doesn't have to provide a default case. The interpreter will error if the `cond` expression isn't exhaustive and no default case was provided, though.

3.3. Let bindings and the scope

In MalbolgeLISP, there is a single way to bind variables without scope (via macros), and two ways of binding lexically scoped variables. Variables are usually bound using `lambda` and all the covers over it (`dyad`, `monad`, `defun`), but there is a way to bind variables without explicitly using a lambda expression - `let` bindings. `let` accepts a list of atoms to be bound interleaved with their values and the code to execute in the freshly created scope. The main difference from `let` present in other lisps is that the binding list is flattened:

```
% (let (x 5 y 6) (print [x + y]))
.....................|............
11

11
```

When evaluated, `let` is desugared to a lambda expression with reversed parameters. The conversion schema follows:

```
(let (x x0 y y0 z z0 ...) (...)) => ((lambda (z y x ...) (...)) z0 y0 x0 ...)
```

The real parameter order is reversed, since the interpreter (for the sake of simplicity) iterates over the input list and uses `cons` to create the resulting desugared expression. Single-variable `let` bindings could illustratively also be implemented as a macro:

```
% (defmacro let' (x y c) ((lambda (x) (eval c)) y)))
.............................|.
macro/syn
% (let' x 5 (print x))
...........|...........
5

5
```

The current scope is bound to the lambda expression at the time of it's creation. Since lazily evaluated values utilise closures which are lexically scoped, this property applies to them as well. Consequently, everything inside the lazily evaluated expression has access to the lexical ancestor's bound variables. Generally speaking, the global scope can be modified using `def`, `defun` and `defmacro`, regardless of where these functions appear. Modifying the local scope is possible using only lambda expressions and the `let` function. It's discouraged to use `def`, `defun` and `defmacro` outside of the top-level code scope (i.e. inside lambda expressions, iterated callables, etc...), since it introduces global and mutable state.

3.4. Lisp-style list processing

MalbolgeLISP supports Lisp-style list processing (usually distinguished by extensive use of primitives mentioned below and recursion), even though the more sophisticated and concise Haskell and APL-like functionality is also supported.

`car` is used to query a list's head (`1∘↑`). It doesn't perform any copies and returns the atom the list's head points to. `car` of an empty list is *NULL*.

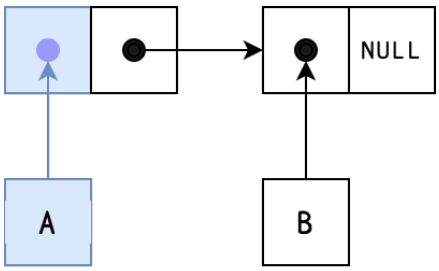

Figure 3.1: The result of car invocation

`cdr` is used to query a list's tail (`1↓`), i.e. everything besides it's head, which is why it's sometimes called *beheading*. `cdr` of a single-element or empty list is *NULL*.

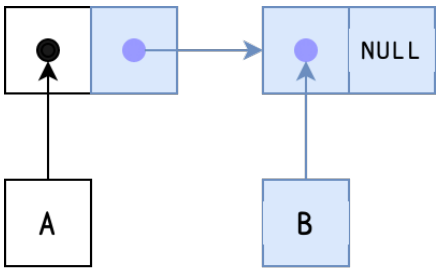

Figure 3.2: The result of car invocation

`cons` is used to prepend a node (head and a tail) to a list. The tail of the newly created node with given head is set to the provided list. For instance, the list demonstrated in previous examples could be made using `(cons A (cons B null))`. A more concise version of this expression is `(tie A B)`. `tie` behaves like a chain of `cons` invocation, where the last invocation prepends to a *NULL* list. A parallel could be drawn between `cons & tie` and `if & cond`.

3.5. Functional list processing

In MalbolgeLISP, the preferred way to process data involves functional devices introduced with MalbolgeLISP v1.2. Most of them were borrowed from Haskell[5] and APL[6].

3.5.1. iota

The `iota` function exhibits the same behavior as it's C++ counterpart[7], although the concept of an index generator has been introduced by APL. `iota` assumes the index origin of 0, so it generates indices in range $[0, n)$. Unlike in APL, `iota` isn't ambivalent nor doesn't support taking a list as it's only argument:

```
; MalbolgeLISP
% (iota 5)
.....|...
(0 1 2 3 4)
% (iota '(3 3))
..........|...E030
% (iota '(1 2 3) '(4 5 6))
..................|..E029
```

[5] `intersperse`, `filter`, `zipwidth`, etc..
[6] to name a few: `where`, `take`, `drop`, `map`, `replicate`, `rev`
[7] `https://en.cppreference.com/w/cpp/algorithm/iota`

```
      ⍝ APL
      ⎕io←0
      ⍳ 3 3
┌───┬───┬───┐
│0 0│0 1│0 2│
├───┼───┼───┤
│1 0│1 1│1 2│
├───┼───┼───┤
│2 0│2 1│2 2│
└───┴───┴───┘
      ⍳ 5
0 1 2 3 4
      'ABCDEF'⍳'ACF'
0 2 5
```

3.5.2. size

The `size` functions yields the length of a list (in $O(n)$ time complexity). It traverses the list shallowly and doesn't account for it's depth. For instance, a very inefficient identity function on numbers could be implemented to demonstrate it's behavior:

```
; MalbolgeLISP
% (def f (atop size iota))
..........|......
bind/syn
% (f 6)
.....|.........
6
% (f 3)
.....|.........
3
```

```
      ⍝ APL
      f←≢⍳
      f 6
6
      f 3
3
```

3.5.3. n-th

Picking arbitrary elements from a list is done using `nth`, or a constant n-th (the `#` prefix). For example, to pick n-th element from the end of a list, the following function might be used:

```
; MalbolgeLISP
% (def nl (dyad (nth [(size y) - [x + 1]] y)))
.............................|..
(lambda (x y) (nth (- (size y) (+ x 1)) y))
% (nl '(1 2 3 4 5) 2)
..............|...............
3
```

```
      ⍝ APL
```

```
      nl←⊃⍵
      2 nl 1 2 3 4 5
3
```

Constant n-th is an alternative way to query the n-th (where n is constant) element of a list. `#N` is equivalent to `(bind nth N)`. *constant n-th* is in many ways similar to a tack - it could even be implemented using a tack and `lift`. For instance:

```
; Demonstration of constant n-th
% (print ((atop #1 map) (bind lazy ~) '(0 1 1 1 1 0)))
.............................|.............................................
0

0

; Correspondence between constant n-th and tack/lift:
#N <=> (bind lift $N)
```

3.5.4. map

`map` is an ubiquitous higher-order function originating from functional languages present in C++[8], APL[9] or Java[10]. `map` is reponsible for transforming (*mapping*) one list into another of equal length, by calling a functor on every element of the original list. It takes any callable first argument, and a list second argument. For example:

```
% (map (bind selfie *) (iota 6))
...............|.....................................................
(0 1 4 9 16 25)
% (map ~ '(1 0))
...........|..........
(0 1)
% (map ~ null)
......|....
null
```

3.5.5. filter

`filter` is a higher order function that conditionally removes elements of a list. First, the list is mapped, and then the elements that correspond to falsy values in the list obtained by mapping are removed, and truthy values are kept.

`filter` behaves in the same way as `map` when the second argument is *NULL*. To demonstrate:

```
; MalbolgeLISP
% (filter (bind' % 2) (iota 10))
...............|.....................................................
(1 3 5 7 9)
% (filter (bind' % 2) null)
...........|..
null
```

[8] https://en.cppreference.com/w/cpp/algorithm/transform

[9] https://help.dyalog.com/latest/index.htm#Language/Primitive%20Operators/Each%20with%20Monadic%20Operand.htm

[10] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map-java.util.function.Function-

```
      ⍝ APL
      filter←{⍵/⍨⍺⍺¨⍵}
      2∘| filter ⍳10
┌→─────────┐
│1 3 5 7 9│
└~─────────┘
      2∘| filter ⍬
┌⊖┐
│0│
└~┘
```

3.5.6. rev

`rev` is an ambivalent built-in function inspired by APL's *reverse/rotate*[11]. The monadic case simply reverses a list, while the dyadic case rotates it. Since MalbolgeLISP doesn't support rotation with a negative argument, but the dyadic rotation with negative parameter can be defined in an alternative way.

```
; MalbolgeLISP
% (rev (iota 5))
.........|.....
(4 3 2 1 0)
% (rev 3 (iota 6))
..........|......
(3 4 5 0 1 2)
% (def rev' (dyad (rev [(size y) - x] y)))
.........................|..
(lambda (x y) (rev (- (size y) x) y))
% (rev' 2 (iota 6))
..........|..............
(4 5 0 1 2 3)
```

```
      ⍝ APL
      ⌽⍳5
┌→─────────┐
│4 3 2 1 0│
└~─────────┘
      3⌽⍳6
┌→───────────┐
│3 4 5 0 1 2│
└~───────────┘
      ¯2⌽⍳6
┌→───────────┐
│4 5 0 1 2 3│
└~───────────┘
```

3.5.7. any, every

`any` and `every` are closely tied to each other. `any` returns `1` if its functor returned a truthy value for **any** element (and `0` otherwise), and `every` returns `1` if its functor returned a truthy value for **every** element (and `0` otherwise). `any` and `every` short-circuit (otherwise, they would be easy to implement using `filter`). To demonstrate:

```
; MalbolgeLISP
```

[11] `https://aplwiki.com/wiki/Reverse` and `https://aplwiki.com/wiki/Rotate`

```
% (every (bind = 2) '(2 2 2 2))
..................|..............................
1
% (any (bind = 2) '(2 2 2 2))
..................|..............................
1
% (every (bind = 3) '(3 3 2))
..................|.............
0
% (any (bind = 3) '(0 1 2))
..................|.............
0
```

```
      ⍝ APL
      ⍝ Note: This implementation doesn't short-circuit
      any←{∨/⍺⍺¨⍵}
      every←{∧/⍺⍺¨⍵}
      2∘= every 2 2 2
1
      2∘= any 1 2 3
1
      3∘= every 3 3 2
0
      3∘= any 1 2 2
0
```

3.5.8. zip, zipwith

`zip` juxtaposes elements from two lists to form a list of pairs of corresponding elements from them. The pairs can be further processed to produce a flat result if `zipwith` is used. `zip` and `zipwith` will return a list of size `⌊⍤≠`[12].

```
; MalbolgeLISP
% [zip selfie (iota 3)]
..........|..........
((0 0) (1 1) (2 2))
% [+ zipwith (iota 3) (iota 3)]
................|.....................
(0 2 4)
```

```
      ⍝ APL
      zip←,¨
      zipwith←{⍺⍺/¨⍺,¨⍵}
      zip⍨ ⍳3
┌→──────────────────┐
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │0 0│ │1 1│ │2 2│ │
│ └~──┘ └~──┘ └~──┘ │
└∊──────────────────┘
      (⍳ 3) (+ zipwith) (⍳ 3)
┌→────┐
│0 2 4│
```

[12] when end of list is encountered when the pairs are formed, the operation finishes

```
└~────┘
```

3.5.9. flatten, flatmap

`flatten` and `flatmap`'s mutual relations somewhat resemble `zip` and `zipwith`. While the first operation is a function that simply flattens a list, the second function is a `map` which result is flattened afterwards. List flattening isn't deep:

```
; MalbolgeLISP
% (flatten '((1 2) (3 4)))
.................|...
(1 2 3 4)
% (flatten '(((1 2) (3 4)) (3 4)))
.........................|...
((1 2) (3 4) 3 4)
; Simple deep flattening
% (def deepf (bind iterate = flatten))
............|........
bind/syn
% (deepf '(((1 2) (3 4)) (3 4)))
.........................|............................
(1 2 3 4 3 4)
; Successor and predecessor of a number
% (def sp (monad (tie [x + 1] [x - 1])))
.........................|..
(lambda (x) (tie (+ x 1) (- x 1)))
% (flatmap sp '(1 2 3 4))
.............|..........................................................
(2 0 3 1 4 2 5 3)
; As opposed to...
% (map sp '(1 2 3 4))
.............|..........................................................
((2 0) (3 1) (4 2) (5 3))

      ⍝ APL
      f←⊃,/
      f ((1 2) (3 4))
1 2 3 4
      f (((1 2) (3 4)) (3 4))
┌─────┬─────┬─┬─┐
│1 2│3 4│3│4│
└─────┴─────┴─┴─┘
      ⍝ two choices for deep flattening
      ⍝ MalbolgeLISP port:
      g←f⍤≡
      ⍝ Idiomatic APL:
      h←∈
      g (((1 2) (3 4)) (3 4))
1 2 3 4 3 4
      h (((1 2) (3 4)) (3 4))
1 2 3 4 3 4
      sp←(+,-)∘1
      fm←{⊃,/⍺⍺¨⍵}
      sp fm 1 2 3 4
```

```
2 0 3 1 4 2 5 3
      sp¨ 1 2 3 4
┌─┬─┬─┬─┬─┬─┬─┬─┐
│2 0│3 1│4 2│5 3│
└─┴─┴─┴─┴─┴─┴─┴─┘
```

Illustratively, flattening a list alters it in the following way:

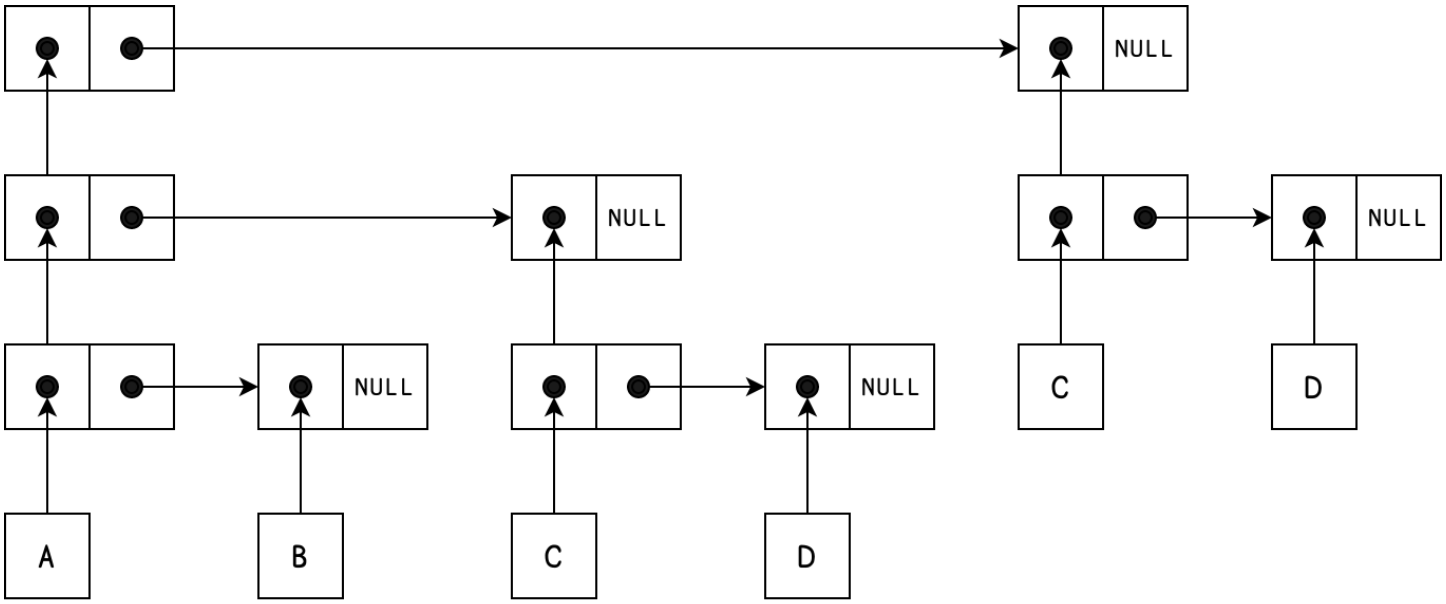

Figure 3.3: A list before flattening

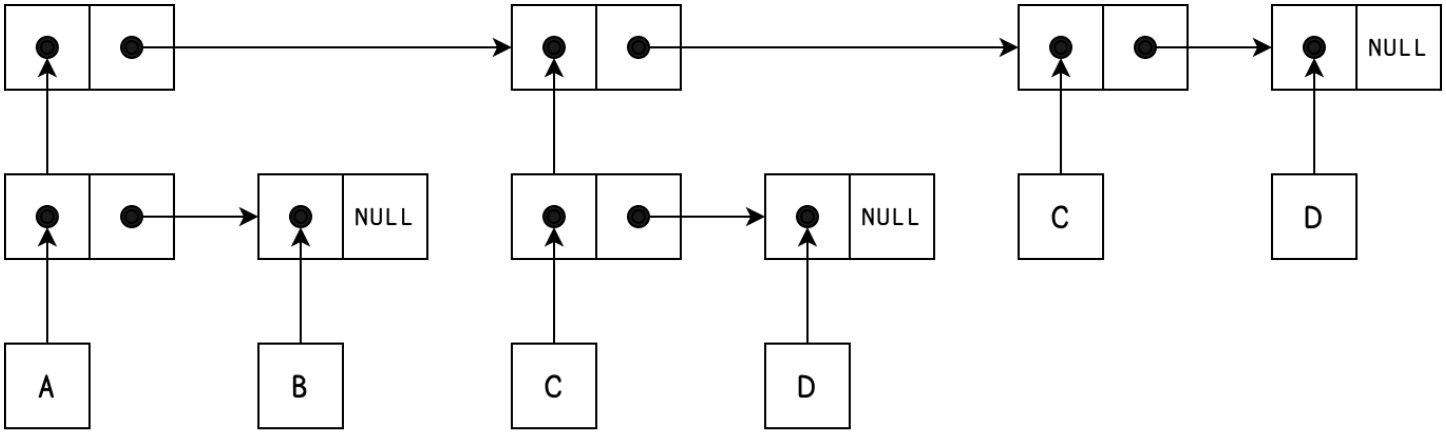

Figure 3.4: A list after flattening

3.5.10. folds

Folding is a way to transform an entire list by putting a binary operation between each element of it. Since folding must yield a result, and the list might be empty, `fold` takes an additional argument which specifies the identity element, which semantically means that the value is left unchanged[13]. Since APL supports only reductions, the comparison between MalbolgeLISP and APL is not demonstrated in this example.

```
% (fold 0 + '(1 2 3 4 5))
```

[13] for instance, the identity element of $+$ is `0`, since $x + 0 = x$ and the identity element of $\times$ is `1`, since $x \times 1 = x$

```
..............|.......................
15
% [[[[[0 + 1] + 2] + 3] + 4] + 5]
........................|.............
15
% (fold 0 + null)
.......|.....
0
```

If it is known that the list contains at least one element, `fold'` can be used, which is equivalent to APL's reductions:

```
; MalbolgeLISP
% (fold' + '(1 2 3 4 5))
..............|.......................
15
```

```
      ⍝ APL
      +/ 1 2 3 4 5
15
```

3.5.11. where

`where` is a function borrowed from APL - ⍸[14]. It returns the indices on which the input array contains truthy values. If the truthy value is greater than one, it's repeated that amount of times. For example:

```
; MalbolgeLISP
% (where '(1 0 1 0 1 1 1 0))
.............|...
(0 2 4 5 6)
% (where '(1 2 3 4))
............|...
(0 1 1 2 2 2 3 3 3 3)
```

```
      ⍝ APL
      ⍸1 0 1 0 1 1 1 0
0 2 4 5 6
      ⍸1 2 3 4
0 1 1 2 2 2 3 3 3 3
```

3.5.12. count

`count` counts the amount of times it's functor returned a truthy value for each element of a list. It can be trivially expressed as a `fold` and `map`. An example of `count` usage follows.

```
; MalbolgeLISP
% (count (bind = 2) '(2 2 3 2 2 3))
.................|.........................................
4
% (def count' (dyad (fold 0 + (map [[< bind 0] atop x] y))))
............................|..
```

[14] http://help.dyalog.com/16.0/Content/Language/Primitive%20Functions/Where.htm

```
(lambda (x y) (fold 0 + (map (atop (bind < 0) x) y)))
% (count' (bind = 2) '(2 3 2 3))
..................|.............................................
..................................................
2
```

```
      ⍝ APL
      count←{+/(0<⍺⍺)¨⍵}
      2∘= count 2 3 2 3 2 2
4
```

3.5.13. replicate

`replicate` is one of the most overloaded functions in MalbolgeLISP. It takes three different forms depending on the argument types:

- `replicate list list` - copy elements from list 2 according to the masks in list 1. comparable to `filter`[15], but it takes a pre-mapped array instead of a functor.
- `replicate num list` - duplicate the list specified amount of times
- `replicate num any` - make a list out of any atom repeated specified amount of times

```
; MalbolgeLISP
% (replicate 5 hello)
......|....
(hello hello hello hello hello)
% (replicate 5 '(1 2))
...........|....
(1 2 1 2 1 2 1 2 1 2)
% (replicate '(1 0 2) '(1 2 3))
.................|....
(1 3 3)
```

```
      ⍝ APL
      replicate1←⍴∘⊂
      5 replicate1 'hello'
┌─────┬─────┬─────┬─────┬─────┐
│hello│hello│hello│hello│hello│
└─────┴─────┴─────┴─────┴─────┘
      replicate2←{↑,/⍺⍴⊂⍵}
      5 replicate2 1 2
1 2 1 2 1 2 1 2 1 2
      1 0 2/1 2 3
1 3 3
```

3.5.14. scan

`scan` performs a `fold` with partial results. The standard behavior diverges from APL (since the identity element is excluded), but it can be accomplished using `scan'`, which assumes that the input list has at least a single element.

```
; MalbolgeLISP
```

[15] a filter-like function is implemented in APL using replicate

```
% (scan' * '(1 2 3 4 5))
..............|.....................
(1 2 6 24 120)
% (scan * 1 '(1 2 3 4 5))
..............|.......................
(1 1 2 6 24 120)
```

```
      ⍝ APL
      ×\1 2 3 4 5
1 2 6 24 120
```

3.5.15. uniq

`uniq` returns unique elements of a list using the *formal definition of equality*. It uses an algorithm which makes $O(n^2)$ equality checks in the pessimistic case. To demonstrate:

```
; MalbolgeLISP
% (uniq '(1 6 2 5 2 6 2 3))
................|...
(1 6 2 5 3)
% (uniq '((1 2) (1 2) (1 2 3) (4 5) 6 (4 5)))
................................|...
((1 2) (1 2 3) (4 5) 6)
```

```
      ⍝ APL
      ∪ 1 6 2 5 2 5 6 2 3
1 6 2 5 3
      ∪ ((1 2) (1 2) (1 2 3) (4 5) 6 (4 5))
┌─────┬─────┬─┐
│1 2│1 2 3│4 5│6│
└─────┴─────┴─┘
```

3.5.16. sort

`sort` is a function that sorts a list with an arbitrary comparator (or assumes a default comparator for sorting numeric lists, if none was provided). MalbolgeLISP utilises the insertion sort algorithm[16]. It has a time complexity of $O(n^2)$ and auxiliary space requirement of $O(1)$. It takes maximum time to sort a list if elements are sorted in reverse order, and it takes minimum time when the elements are already sorted. For example:

```
; MalbolgeLISP
% (sort '(1 9 5 8 2 3 9 5))
................|...
(1 2 3 5 5 8 9 9)
% (sort '(1 9 5 8 2 3 9 5))
................|...
(1 2 3 5 5 8 9 9)
% (sort > '(1 9 5 8 2 3 9 5))
................|.....................................
................................................
.....................
(1 2 3 5 5 8 9 9)
```

[16]because of it's simplicity and performance on small lists, which are going to realistically be the main use case for it in MalbolgeLISP

```
      ⍝ APL
      ⍝ Implementation of an insertion sort.
      sortn←((≥⊢⍤⊃/⊢),⊣,<⊢⍤⊃/⊢)/
      sortn 32 4 1 34 95 3 2 120 ¯38
┌──────────────────────────┐
│¯38 1 2 3 4 32 34 95 120│
└──────────────────────────┘
      ⍝ A more idiomatic way of solving the problem:
      sortn1←{⍵[⍋⍵]}
      sortn1 32 4 1 34 95 3 2 120 ¯38
¯38 1 2 3 4 32 34 95 120
      ⍝ Sorting with a comparator
      sortc←{p←⍺⍺⋄{r←⍵/⍨⍵⋄c←⍺p⍵⋄(r c),⍺,r ~c}/⍵}
      > sortc 32 4 1 34 95 3 2 120 ¯38
┌──────────────────────────┐
│¯38 1 2 3 4 32 34 95 120│
└──────────────────────────┘
```

3.5.17. take, take', drop, drop'

`take` and `take'` extract n elements from the front (`take`) or from the back (`take'`) of a list. They correspond to `↑` and `((-⊣)↑⊢)` in APL. `take` triggers a copy, while `take'` doesn't:

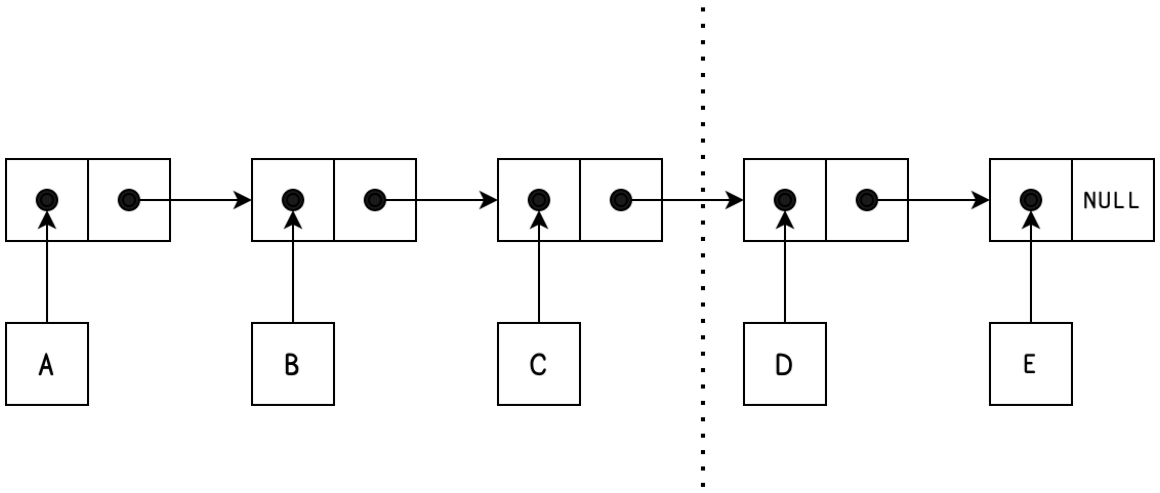

Figure 3.5: Taking the first three elements from a list

The list must be cloned, since setting the tail of the list node which holds `C` to *NULL* (to yield a new list) would invalidate existing references. This doesn't apply to `take'`, since it doesn't have to modify the memory it's operating on.

```
; MalbolgeLISP
% (take 3 '(1 2 3 4 5))
..............|....
(1 2 3)
% (take' 3 '(1 2 3 4 5))
..............|....
(3 4 5)
```

```
      ⍝ APL
      3↑1 2 3 4 5
1 2 3
      ¯3↑1 2 3 4 5
3 4 5
```

`drop` and `drop'` exhibit the same behavior. They drop n elements from the front (`drop`) or the back (`drop'`) of a list. Dropping elements from the back requires a copy, while dropping them from the front does, for the reasons outlined above.

```
; MalbolgeLISP
% (drop 3 '(1 2 3 4 5))
..............|....
(4 5)
% (drop' 3 '(1 2 3 4 5))
..............|....
(1 2)
```

```
      ⍝ APL
      3↓1 2 3 4 5
4 5
      ¯3↓1 2 3 4 5
1 2
```

3.6. Iteration and recursion

This section is intended to demonstrate various implementations of the Fibonacci series in MalbolgeLISP[17] using iteration and recursion. Then, the approaches will be judged by cleaniness, conciseness and performance.

The naive, doubly recursive attempt timed at 1m 19s follows:

```
(defun fib1 (n) (
    if [n < 2]
        n
        [(fib1 [n - 1]) + (fib1 [n - 2])]))
(fib1 6) ; => 8
; APL: {1≥⍵:⍵ ⋄ (∇⍵-2)+∇⍵-1} 6
```

A singly-recursive attempt which keeps track of the accumulator tuple. There exist two versions of it - a port of the APL solution and the idiomatic MalbolgeLISP attempt, which is much faster than the port, since it takes advantage of MalbolgeLISP's support of functions of arity 3 or higher.

```
; APL version: {α←0 1 ⋄ 0=⍵:⊃α ⋄ (1↓α,+/α)∇⍵-1} 6
; Direct port at 1m 6s:
(defun fib2 (n) ((lambda (a w) (
    if [w = 0]
        (#0 a)
        ((bruijn 0) (tie (#1 a) (lift + a)) [w - 1]))) '(0 1) n))
; A more idiomatic solution at 54s:
(defun fib2 (n) ((lambda (x y w) (
    if [w = 0]
        x
        ((bruijn 0) y [x + y] [w - 1]))) 0 1 n))
```

[17] in comparison to APL

An iterative attempt timed at 43s:

```
; APL version: {⊃+\∘⍵⍣⍵⍳2} 6
(defun fib3 (n) (#0 (
    iterateN n (lambda (x) (
        tie (#1 x) [(#0 x) + (#1 x)])) '(0 1))))
```

It should be noted that even though the second approach is generally faster, it's not as clean as the first or third one. The first approach is the slowest, while the second and third approaches are faster. Generally, the most concise, clean and the fastest attempt is the iterative attempt.

Chapter 4

Summary

MalbolgeLISP is an enormously big project. At 21'000 lines of pre-processed asm2bf code, 380'000 lines of low level assembly code and 368'000'000 bytes of Malbolge code, it's the largest project I ever worked on. I feel that I managed to accomplish my goal of creating the most complex Malbolge Unshackled program to date, while making it usable on contemporary mid-end PCs. I put months of research into Malbolge, and in total, I spent around two full months writing MalbolgeLISP, documenting it, optimising it, and polishing the Malbolge toolchain.

The Malbolge toolchain 57'000 lines of code, featuring C, C++, APL, Perl, Python, Lua, and many others. It's complexity grew over time, and in conjunction with asm2bf, I believe that it's the most complex project I have ever worked on. While developing and documenting it, I have:

- improved my APL programming abilities.
- introduced myself to SAT solvers.
- stepped up my cryptography skills.
- learned how to implement functional languages in restricted environments.
- learned about many concepts in functional programming and mathematics that i didn't know about before.
- had an occasion to truly understand Lisp.
- improved at interpreter development.
- became better at point-free programming.
- managed to make one of my projects gain traction.
- done something new. Explored something that was yet unexplored. *I was the best.*

Finally, I'd like to thank my girlfriend, Kiera, for giving me motivation to work and improve. She is someone i desperately needed in my life.

Appendix A - compression benchmark data

To process the results of compression benchmarks (compute compression ratio, median and average speed of compression and decompression), the following program is used:

```
⍝ The data is expected to be supplied in an interleaving format
⍝ between compression and decompression timings (s) as the argument ⍵.
⍝ The vector of original and compressed size is supplied as the argument ⍺.
 parse←{
     ⍝ compute the compression ratio.
     cr←÷/⍺
     ⍝ uninterleave the data.
     sh←2,¨2÷¨⍴⍵⋄d←sh⍴⍵
     ⍝ average and median
     avg←(+⌿÷≢)d
     med←(2÷¨⍨1⊥⊢⌷¨⍨∘⍋⌷¨⍨∘⍋⌈2÷¨⍨0 1+≢)¨↓⍉d
     ⍝ display results
     ⎕←⍝'Compression ratio:    ' (⍕cr)
     ⎕←⍝'Avg pack/unpack time: ' (⍕avg)
     ⎕←⍝'Med pack/unpack time: ' (⍕med)
 }
```

Since some results are yielded by the `time` UNIX utility in the `mm:ss.uu` format, instead of the ready to use `ss.uu` format, the following function automatically performs the conversion:

```
 fixmin←{
     s←≠⊂⊢⋄d←' 's ⍵⋄cn←(⊃∘⍋⌷⍨⎕VFI)¨
     ms←':'∘⍝¨d⋄f←{60⊥⍎cn':'s ⍵}¨
     fM←f@(⍸ms)⋄fS←cn@(⍸~ms)⋄⍎,/fS fM d
 }
```

```
      339823649 15349140 parse fixmin ppmd5Data ⍝ PPMD -mx=5
Compression ratio:    22.13958886
Avg pack/unpack time: 10.7432 13.4947
Med pack/unpack time: 10.796 13.345
      339823649 6029994 parse fixmin ppmd9Data  ⍝ PPMD -mx=9
Compression ratio:    56.35555342
Avg pack/unpack time: 23.591 25.5343
Med pack/unpack time: 23.5785 25.426
      339823649 37005733 parse fixmin deflate5Data  ⍝ Deflate -mx=5
Compression ratio:    9.183000077
Avg pack/unpack time: 64.624 2.217
Med pack/unpack time: 64.64 2.1185
      339823649 35905627 parse fixmin deflate9Data  ⍝ Deflate -mx=9
Compression ratio:    9.464356353
Avg pack/unpack time: 443.03 2.4009
Med pack/unpack time: 441.985 2.396
```

```
      339823649 12281576 parse fixmin bzip25Data  A BZip2 -mx=5
Compression ratio:    27.66938453
Avg pack/unpack time: 4.0545 10.3362
Med pack/unpack time: 3.972 10.251
      339823649 12266587 parse fixmin bzip29Data  A BZip2 -mx=9
Compression ratio:    27.70319478
Avg pack/unpack time: 19.0515 9.8867
Med pack/unpack time: 19.111 9.761
      339823649 14622648 parse fixmin lzma5Data  A LZMA -mx=5
Compression ratio:    23.23954245
Avg pack/unpack time: 79.17 2.2699
Med pack/unpack time: 79.275 2.1905
      339823649 12067455 parse fixmin lzma9Data  A LZMA -mx=9
Compression ratio:    28.1603411
Avg pack/unpack time: 177.241 2.2187
Med pack/unpack time: 177 2.145
      339823649 37671991 parse gzip6Data  A gzip -6
Compression ratio:    9.020591691
Avg pack/unpack time: 12.2658 1.4981
Med pack/unpack time: 12.2315 1.4975
      339823649 37554328 parse gzip9Data  A gzip -9
Compression ratio:    9.048854476
Avg pack/unpack time: 14.9222 1.5072
Med pack/unpack time: 14.8975 1.4805

PPMD -mx=5: 10.471 13.846 10.381 13.524 10.873 13.358
            11.109 13.062 11.044 13.231 10.677 13.332
            10.734 14.142 10.858 14.063 10.380 13.228
            10.905 13.161
PPMD -mx=9: 22.564 27.833 22.839 26.289 20.872 26.227
            23.976 25.358 24.987 25.863 23.751 24.408
            23.406 24.874 25.510 25.494 23.265 23.815
            24.740 25.182
Deflate -mx=5: 1:04.76 2.359 1:04.68 1.911 1:04.29 1.942
               1:04.16 2.788 1:04.87 2.128 1:04.60 2.700
               1:04.78 2.038 1:04.37 1.962 1:04.30 2.109
               1:05.43 2.233
Deflate -mx=9: 7:20.82 2.547 7:20.39 2.934 7:21.92 2.714
               7:22.05 2.797 7:29.80 1.969 7:21.01 2.366
               7:24.64 2.310 7:25.06 1.996 7:23.13 2.426
               7:21.48 1.950
BZip2 -mx=5: 5.145 10.322 4.296 10.578 4.113 11.341
             4.156 10.180 4.155 9.944 3.707 10.636
             3.733 10.425 3.685 9.950 3.831 10.140
             3.724 9.846
BZip2 -mx=9: 18.444 10.030 19.329 9.848 19.300 10.159
             18.855 11.023 18.933 10.286 19.252 9.582
             19.372 9.397 18.808 9.674 18.989 9.293
             19.233 9.575
LZMA -mx=5: 1:20.08 2.036 1:19.11 2.038 1:19.51 2.080
            1:19.11 2.152 1:19.58 2.781 1:17.84 2.229
            1:20.13 2.412 1:18.90 2.401 1:18.00 2.425
            1:19.44 2.145
LZMA -mx=9: 3:00.12 2.118 2:56.99 2.040 2:55.21 2.041
```

```
            2:57.16 2.221 2:56.97 2.063 2:57.01 2.172
            2:59.83 2.556 2:54.96 2.604 2:55.01 2.118
            2:59.15 2.254
GZip -6: 12.729 1.502 12.355 1.515 12.316 1.501
         12.048 1.489 12.230 1.490 12.220 1.487
         12.076 1.494 12.357 1.507 12.233 1.493
         12.094 1.503
GZip -9: 15.001 1.491 14.908 1.518 15.195 1.472
         15.109 1.480 14.704 1.710 14.706 1.477
         14.887 1.477 14.861 1.478 14.974 1.481
         14.877 1.488
```

Appendix B - auxiliary code

The lazy evaluation chapter's footnote 25 mentions a C++ implementation of suspended functions, which is outlined below.

```
#include <functional>
#define _(a...) {return({a;});}

template<class T>
class suspension {
private:
    T const & (*t)(suspension *);
    mutable T m;
    std::function<T()> f;

    T const & gM() _ ( m )
    T const & sM() _ ( m = f(); t = [] (suspension * x) _ ( x->gM() ); m )
public:
    explicit suspension(std::function<T()> f)
        : f(f), t([] (suspension * x) _ ( x->sM() )), m(T()) { }
    T const & get() _ ( t(this) )
};
```

Since MalbolgeLISP doesn't have a type system, implementing some features present in functional languages might be unintuitive and the resulting code is brittle and unchecked. That being said, it's possible to implement some features like the Maybe monad[1]:

```
(defun Just (x) (tie 1 x))
(defun Nothing (x) (tie 0 0))
(defun unwrap (x) (if (= 0 (car x)) (off) (car (cdr x))))
```

Unwrapping an empty optional turns off the interpreter. To demonstrate:

```
(Just 1)
......|.......
(1 1)
(unwrap (Just 1))
..........|.....................
1
(unwrap (Nothing))
........|.... (*interpreter terminates*)
```

[1] borrowed from Haskell, although it's present in other languages as an optional type; the implementation was suggested by Matt8898

Appendix D - Reimplementation of sorting

The following sorting functions that operate on numbers have been suggested[2] as demonstration programs for MalbolgeLISP v1.1:

```
(defun filter (f l) (cond
    ((= l null) null)
    ((f (car l)) (cons (car l) (filter f (cdr l))))
    (1 (filter f (cdr l)))))

(defun append (a b) (if (= null a) b (cons (car a) (append (cdr a) b) )))
(defun append3 (a b c) (append a (append b c)))
(defun list>= (m list) (filter (lambda (n) (! (< n m))) list))
(defun list< (m list) (filter (lambda (n) (< n m)) list))

(defun qsort (l) (
    if (= null l)
        null
        (append3
            (qsort (list< (car l) (cdr l)))
            (cons (car l) null)
            (qsort (list>= (car l) (cdr l))))))

(qsort '(6 8 1 0 6 8 2))

; -------------

(defun insert (i l) (
    if (= null l)
        (cons i l)
        (if (> i (car l))
            (cons (car l) (insert i (cdr l)))
            (cons i l))))

(defun insertsort (l) (
    if (= null l)
        null
        (insert (car l) (insertsort (cdr l)))))

(insertsort '(6 8 1 0 6 8 2))
```

The second algorithm is faster than the first one, even though the first one has lower computational complexity. This is related to the code size (the first implementation is being parsed for much longer than the first one) and the complexity (insertion sort works well for small vectors).

[2] by Matt8898 and Umnikos; Matt8898 have admitted themselves that the ”quicksort” implementation doesn’t implement a quicksort