Lucas S. Vieira

Believe

A Bel interpreter built in C

Version 0.3 (Alpha)

April 18, 2021

Contents

Acknowledgements **1**

1 Introduction **3**
1.1 About literate programming . . . 3
1.2 Licensing . . . 4
Textbook license . . . 4
Code license . . . 5
1.3 Contribution guidelines . . . 5
Code contribution guidelines . . . 5
Project communication guidelines . . . 6
1.4 Backlog . . . 8
Roadmap . . . 8
On-the-fly checklist . . . 10

2 Tools and scripts **11**
2.1 Makefile . . . 11
2.2 Memory leak testing . . . 11
2.3 Tangling . . . 12
2.4 Running the program . . . 12

3 Libraries and headers **13**
3.1 File header . . . 13
3.2 Software-related definitions . . . 13
3.3 Default headers . . . 14
3.4 Boehm-Demers-Weiser Garbage Collector . . . 14

4 Fundamental data types **15**
4.1 Enumerating Bel types . . . 15
4.2 Pair . . . 15
4.3 Character . . . 16
4.4 Symbol . . . 16
4.5 Stream . . . 16
4.6 Number . . . 17
4.7 The `Bel` structure . . . 18

5 Essential structures and manipulation of data **19**
5.1 Basic definitions . . . 19
Forward declarations . . . 20
5.2 Predicates . . . 20
`symbolp` . . . 20
`nilp` . . . 20
`pairp` . . . 21
`atomp` . . . 21
`charp` . . . 21
`streamp` . . . 21
`numberp` . . . 21
`idp` . . . 21
`errorp` . . . 23
`proper-list-p` . . . 23
`stringp` . . . 24
`literalp` . . . 25
`primitivep` . . . 25
`closurep` . . . 25
`quotep` . . . 25
`number-list-p` . . . 26
5.3 Symbol Table and Symbols . . . 26
5.4 Pairs . . . 28
5.5 Characters and Strings . . . 31
5.6 Streams . . . 34
Stream manipulation safety . . . 38
5.7 Numbers . . . 39
Number generation . . . 39
Number arithmetic . . . 41
5.8 Errors . . . 52

6 Axioms **55**
6.1 Variables and constants . . . 55
6.2 List of all characters . . . 55
6.3 Environment . . . 57
Types and hierarchy of environments . . . 62
Environment extension and capturing . . . 62
6.4 Literals . . . 63
Primitives . . . 64
Closures . . . 66

7 Printing **67**
7.1 Forward declarations . . . 67
7.2 Printing pairs . . . 67
Printing functions . . . 68
7.3 Printing strings . . . 69

7.4 Printing streams . . . 70
7.5 Printing numbers . . . 70
7.6 Generic printing . . . 71

8 Evaluator **73**
8.1 Forward declarations . . . 73
8.2 The *eval* function . . . 74
8.3 The *apply* function . . . 76
8.4 Auxiliary functions . . . 77
Evaluating special forms . . . 77
Evaluate a list of values . . . 82
Apply a primitive operator to a list . . . 82
Bind a list of variables to values . . . 98

9 Reader **101**
9.1 Tokenizer . . . 102
Reserved symbols . . . 103
Read macros . . . 103
Token sizes . . . 104
Tokenization . . . 105
9.2 Parsing . . . 107
Forward declarations . . . 108
Token list parser . . . 108
Token parser . . . 110

10 REPL **113**
10.1 Reading . . . 113
10.2 Evaluation . . . 113
10.3 Printing . . . 113
10.4 Loop . . . 113

11 Debug **115**
11.1 Tests . . . 115
String manipulation and printing . . . 115
List/pair/dotted list notation . . . 115
Proper list notation . . . 116
Closure representation . . . 116
Character list printing and environment lookup . . . 117
Read file bit by bit . . . 118
Display errors . . . 119
Lookup primitives . . . 120
Environment tests . . . 121
Number test . . . 123
Debriefing macro . . . 125
Evaluator test . . . 126

Arithmetic evaluation test 129
Arity tests 131
Dynamic binding test 131
Global binding test 132
Basic tokenizer test 133
Basic parser test 134
Arbitrary input parsing 135
Test-only REPL 136

12 Entry point **139**
12.1 Initialization 139
12.2 Tests 140
12.3 `main` function 141

Acknowledgements

This is an open-source project which anyone can contribute to. I'd like to thank the people who helped me so far with this project.

Many thanks to Carl Mäsak (github.com/masak) not only for contributing with code, but also for highlighting a lot of important aspects in the Bel specification, and also for taking time to discuss other implementation aspects of the interpreter. This kind of contribution is priceless, since it is easy to overlook important details on technical documents. An extra pair of eyes on that regard is always welcome.

CHAPTER 1

Introduction

The goal of this project is to provide a fully-functioning implementation of the Bel language, proposed by Paul Graham. The main goal is not to provide performance; instead, it is supposed to be a didatic approach to implementing a Lisp interpreter.

The code here contained is also a study on how to build a Lisp interpreter from scratch in C. Given that Bel is so simple and is supposed to be a formalism before a commercial language, it seems like the perfect didatic resource to do so.

Here are some useful links with language resources:

- Paul Graham's Bel release website
- Language Guide
- Language Source Code, written in Bel itself
- Bel examples

Note that **this software was a work-in-progress before archiving. Do not expect it to work fully.**

1.1 About literate programming

This interpreter is built using Org with Org-mode in Emacs. Its website specifies that Org is "a format for keeping notes, maintaining TODO lists, planning projects, and authoring documents with a fast and effective plain-text system".

All the code here appears in the order it is written on the actual code files. By using Donald Knuth's concept of literate programming, the relevant code blocks are *tangled* and written in their specified code files, and then the application can be compiled.

By using this approach, I hope to maintain an application where the understanding of what is being written comes before the code itself, so that the reader is able to take and analyse parts of said code based on the prose that accompanies it.

1.2 Licensing

The Believe project is composed of two relevant documents: one being the *textbook*, which contains all the prose parts plus the code blocks in relevant places; and another being the *code*, which is composed solely of the code blocks contained in this *textbook*, and can be understood both as the *code* blocks of the *textbook* or as a separate, *tangled* file containing the relevant discussed code.

When redistributing the *textbook*, one should take the *textbook* license into consideration. But anyone using the *code* parts of the *textbook* or the *tangled* code file included in the project's repository, for *any* purpose, should take the *code* license into consideration as well.

Textbook license

[1] You must provide the name of the creator and attribution parties, a copyright notice, a license notice, a disclaimer notice, and a link to the original material.

[2] You must indicate if you modified the material and retain an indication of previous modifications.

[3] You can see a list of compatible licenses at <https://creativecommons.org/compatible-licenses>.

[4] <https://creativecommons.org/licenses/by-sa/4.0/>

Code license

1.3 Contribution guidelines

Below are described a few guidelines to contribute to this project. It is important to follow them to avoid confusion when contributing.

Code contribution guidelines

1. *This is a literate program.*
   The meaning of this statement is that the software is primarily written as a prose with code blocks. So the way to make an addition to Believe is to open a section on the relevant part and add a code block with a proper code explanation in prose.

2. *The code is written as a C code file.*
   Any new code should be added in a proper C code block. In the end, all C code blocks can be tangled into a single `believe.c` file.

3. *The code uses indentation similar to K&R.*
   Therefore we put types one line above the rest of the function declaration (except for prototypes), and we put the function declaration braces on *the next line*. As for code inside function scopes, the braces should be in front of the block declaration, separated by a single space.

4. *Use `snake_case` and respect naming conventions.*
   All definitions use snake case with lowercase names (e.g. `bel_car`), except for *enumerations* (where the name of the `enum` and its definitions must be uppercase

– e.g. `BEL_SYMBOL`) and *structs* (the default `struct` definition must be uppercase, while a typedef for the `struct` must be lowercase with the first letter in uppercase – e.g. `Bel_pair`).

5. *Indentation must be done with spaces, and each indentation level takes four spaces.*
   Though this is mostly arbitrary, this is the way most of the software was written. This rule is invalid for *Makefiles* only, where the use of *tabs* is relevant.

6. *Do not change the code file directly.*
   Files such as `believe.c` are automatically generated by tangling the code blocks in this file. When creating *pull requests*, it is also desired that the `believe.c` file is not commited with the rest of the changes.

7. *Code without a relevant prose explaining the rationale of what was programmed will not be accepted as contribution.*
   The rationale behind this project is partly related to providing a didatic implementation of a Lisp interpreter, which can be read as a book and implemented by anyone.

These contribution guidelines are subject to change at any time during the software development, as any necessity to clear up confusion appears.

Project communication guidelines

This project does not have a code of conduct. Instead, we rely on the spirit of joyful creation of the participants and on the understanding and cordiality of the people taking part on the project.

By contributing, you understand that the project maintainers are in no way responsible for the misconduct of any participants outside of the scope of this project. In addition, any contributions will be discussed in the light of good faith, not taking into consideration personal aspects such as race, skin color, gender, political views, or any other aspect which is unrelated to the produced code itself. Having said that, any harrassment will not be tolerated, as it is out of the scope of the project; the maintainers of the project will handle the situation in the best possible manner they can.

The maintainers may also remove any other *off-topic* discussions which are completely unrelated to the subject, to avoid pollution on issues, pull requests and such.

It is also important to state that, by contributing, you will also be relying on the good faith and sensibility of the project maintainers to handle the topics above described; again, these are stated merely as a general mentality of moderation, and not as fixed rules which could be circumvented.

To increment this general mentality, we also follow the GNU Kind Communication Guidelines[5], which are not a code of conduct nor a list of rules as well, but more of a reference on how one could approach the discussions and contributions on this software.

[5] <https://www.gnu.org/philosophy/kind-communication.html>

These guidelines are reproduced below in an adapted fashion, with a few changes to accomodate to this project.

1. Please assume other participants are posting in good faith, even if you disagree with what they say. When people present code or text as their own work, please accept it as their work. Please do not criticize people for wrongs that you only speculate they may have done; stick to what they actually say and actually do.

2. Please think about how to treat other participants with respect, especially when you disagree with them. For instance, call them by the names they use, and honor their preferences about their gender identity[6].

3. Please do not take a harsh tone towards other participants, and especially don't make personal attacks against them. Go out of your way to show that you are criticizing a statement, not a person.

4. Please recognize that criticism of your statements is not a personal attack on you. If you feel that someone has attacked you, or offended your personal dignity, please don't hit back with another personal attack. That tends to start a vicious circle of escalating verbal aggression. A private response, politely stating your feelings as feelings, and asking for peace, may calm things down. Write it, set it aside for hours or a day, revise it to remove the anger, and only then send it.

5. Please avoid statements about the presumed typical desires, capabilities or actions of some demographic group. They can offend people in that group, and they are always off-topic in Believe discussions.

6. Please be especially kind to other contributors when saying they made a mistake. Programming means making lots of mistakes, and we all do so - this is why regression tests are useful. Conscientious programmers make mistakes, and then fix them. It is helpful to show contributors that being imperfect is normal, so we don't hold it against them, and that we appreciate their imperfect contributions though we hope they follow through by fixing any problems in them.

7. Likewise, be kind when pointing out to other contributors that they should stop using certain software. We welcome their contributions to our software even if they don't do that. So these reminders should be gentle and not too frequent - don't nag.

8. Please respond to what people actually said, not to exaggerations of their views. Your criticism will not be constructive if it is aimed at a target other than their real views.

[6]Please see the GNU Kind Communication Guidelines for more information on better usage of pronouns and such. Maintainers will not be *enforcing* the usage of particular pronouns, but any misuse of language for blatant purpose of offense will not be tolerated, as it can easily take a discussion to an *off-topic* argument. Finally, always assume that any confusion about pronoun usage from the participants was commited with no offending intention as well, and let the maintainers handle the situation if necessary.

9. If in a discussion someone brings up a tangent to the topic at hand, please keep the discussion on track by focusing on the current topic rather than the tangent. This is not to say that the tangent is bad, or not interesting to discuss - only that it shouldn't interfere with discussion of the issue at hand. In most cases, it is also off-topic, so those interested ought to discuss it somewhere else.
   If you think the tangent is an important and pertinent issue, please bring it up as a separate discussion, if it applies to Believe development.

10. Rather than trying to have the last word, look for the times when there is no need to reply, perhaps because you already made the relevant point clear enough. If you know something about the game of Go, this analogy might clarify that: when the other player's move is not strong enough to require a direct response, it is advantageous to give it none and instead move elsewhere.

11. Please don't argue unceasingly for your preferred course of action when a decision for some other course has already been made. That tends to block the activity's progress.

12. If other participants complain about the way you express your ideas, please make an effort to cater to them. You can find ways to express the same points while making others more comfortable. You are more likely to persuade others if you don't arouse ire about secondary things.

13. Please don't raise *any* political issues in Believe discussions, because they are off-topic.

These communication guidelines are subject to change at any time during the software development, as any necessity to clear up confusion appears.

1.4 Backlog

This is some sort of backlog I once maintained to keep track of what I needed to do and what was already done. There are infinitely better ways to do that, and I'm going to keep it here so that anyone can pick it up where I stopped, if one wishes to.

Roadmap

1. **DONE** Data types
   - Symbols
   - Pairs
     - Lists
   - Characters
   - Strings

- Numbers (opaque; general type only)
  - Integers
  - Float
  - Fractions
  - Complex

2. **DONE** Literals
   - Primitives (representation)
3. **TODO** Environments
   - Environment hierarchy (dyn > lex > glo) [implemented on lookup]
   - Dynamic binding (visible everywhere for short time)
   - Hierarchical lookup (`bel_lookup`)
4. **TODO** Functions
   - Lexical bindings
5. **TODO** Evaluation [wip]
   - Eval [wip]
     - ☐ Special forms [wip]
       - `quote`
       - `lit`
         - `fn` => literal closure
       - `if`
       - ☐ `apply`
       - `join`
       - ☐ `where` => not so straightforward
       - `dyn`
       - ☐ `after`
       - ☐ `ccc` => Later?
       - ☐ `thread` => Later?
       - `set` => global binding
6. Apply

On-the-fly checklist

- Environment functions
- Global environment object `globe`
- Various necessary predicates
  - `stringp` predicate
- Error object
- String printing
- Test for errors on core functions
- Dynamic environment
- Assignments/Unassignments
- `err` primitive function, basic error handling
- Use `BEL_DEBUG` flag everywhere!
- Move debug printing functions to actual printing behaviour
- Prototype evaluator

☐ Proper error propagation

☐ Add proper references with Org-ref and bibtex

☐ Lexical environment object `scope`, shadowable, not unique (is it necessary?)

☐ Prevent circular printing. Particularly useful for environments and closures

CHAPTER 2

Tools and scripts

2.1 Makefile

This software was primarily developed on Void Linux $x86_{64}$, using the Clang compiler. The following Makefile is the one used for building Believe.

```
1  CC     = clang
2  CFLAGS = -std=c17 -g -O2 -Wall -DBEL_DEBUG
3  CLIBS  = -lgc -lm
4  BIN    = believe
5  OBJ    = believe.o
6
7  .PHONY: clean
8
9  $(BIN): $(OBJ)
10         $(CC) $(CFLAGS) $(CLIBS) -o $@ $^
11
12 %.o: %.c
13         $(CC) $(CFLAGS) -c -o $@ $^
14
15 clean:
16         rm -rf *.o $(BIN)
```

2.2 Memory leak testing

This script generates a log file with memory leak information using Valgrind. Valgrind's output is stored in `believe.log`.

```
valgrind --check_leaks=full --log-file="believe.log" -v
↪  ./believe
```

2.3 Tangling

The following snippet can be run from Emacs to enable tangling on save for this file only.

Tangling is the process of taking each block of code and adding it to its specific file. Believe's code will be written in C source files; the Makefile will be written in its own file; and so on. Notice that some blocks (like this one) is not written anywhere, and is meant to be evaluated from inside Emacs.

2.4 Running the program

This script attempts to build and run the Bel interpreter. It will also enable verbose output for the garbage collector.

```
make
export GC_PRINT_STATS=1
./believe
```

CHAPTER 3

Libraries and headers

3.1 File header

Let's add a modest copyright notice to the program's header.

```
/* Believe v0.3                                                 *
 * A Bel Lisp interpreter.                                      *
 * Copyright (c) 2019-2021 Lucas Vieira.                        *
 * This program is distributed under the MIT License. See       *
 * the LICENSE file for details.                                *
 *                                                              *
 * Development information can also be consulted on the         *
 * book which accompanies this software, which was              *
 * written in literate programming form. For more               *
 * information, see https://github.com/luksamuk/believe.        */
```

3.2 Software-related definitions

These definitions relate to program metadata which is going to be displayed on its startup.

```
#define BELIEVE_VERSION   "0.3"
#define BELIEVE_COPYRIGHT "2019-2021 Lucas Vieira"
#define BELIEVE_LICENSE   "MIT"
#define BELIEVE_BUILD_TIME __DATE__ " " __TIME__
```

We'll use a flag for debug which influences the building process. Let's call this flag `BEL_DEBUG`.

When building, if you pass this flag to Clang (see the Makefile), some debug outputs will be available.

By default we'll leave it on, at least for now.

3.3 Default headers

We'll be using `stdio.h` for default console I/O, plus `stdint.h` for some standard integer types. `string.h` provides definitions to handle string manipulation on the C side, however Bel is supposed to have its own string representation, to be discussed later. `errno.h` is used to fetch error strings from streams, for example; and `math.h` is useful for math operations. `stdarg.h` is used for creating variadic functions, and finally, `ctype.h` is used for comparing characters when parsing expressions.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <stdint.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <math.h>
8 #include <stdarg.h>
9 #include <ctype.h>
```

3.4 Boehm-Demers-Weiser Garbage Collector

We also use Boehm-Demers-Weiser GC for garbage collection, instead of programming our own. The `GC_DEBUG` flag helps on debugging. See that we use `-lgc` on the Makefile to link the relevant library to the application.

```
1 #ifdef BEL_DEBUG
2 #define GC_DEBUG
3 #endif
4
5 #include <gc.h>
```

Plus, one could check the Boehm-Demers-Weiser GC tutorial slides by Hans-J. Boehm, for a quick overview of this library.

CHAPTER 4

Fundamental data types

The first thing to do is specify how the data is going to be handled by the interpreter. Here, we define each of these kinds of data. Any procedure for data manipulation will be defined afterwards.

4.1 Enumerating Bel types

We begin by specifying all data types, which Bel has four: symbols, pairs, characters and streams. We also add a number type, which is non-standard, but will be useful; this decision will be explained in its section.

```
1 typedef enum
2 {
3     BEL_SYMBOL,
4     BEL_PAIR,
5     BEL_CHAR,
6     BEL_STREAM,
7     BEL_NUMBER
8 } BEL_TYPE;
```

4.2 Pair

A pair can have two things, which can in return be one of the four data types themselves. Since this is sort of a recursive definition, we need to make a forward declaration of the general `Bel` type, which encloses all four data types.

```
1 typedef struct BEL Bel; // Forward declaration
2
```

```
3 typedef struct
4 {
5     Bel *car;
6     Bel *cdr;
7 } Bel_pair;
```

4.3 Character

A character is nothing but an integer in standard C. For now we'll support only ASCII, so there is no actual need to instantiate the proposed table of characters – this might change in the future.

We say that a character is nothing but a signed 8-bit integer. Should be enough for now.

```
1 typedef int8_t Bel_char;
```

4.4 Symbol

A symbol is nothing but a specific index on the symbol table, so that's how we'll define it.

```
1 typedef uint64_t Bel_sym;
```

4.5 Stream

The stream type is somewhat implementation-dependent. In C, a standard way to refer to streams is to use a `FILE` pointer, since `stdout` and `stdin` themselves are of such type. So we just wrap these pointers in a stream type.

Plus, as per Bel's specification, a stream has three statuses: closed, open for reading, open for writing. Therefore, we use a single enumeration to represent these three states.

Since Bel's specification writes to a stream bit by bit, we need to cache the currently filled byte inside the structure, from left to right, dumping each byte as it is filled. Upon closing, the stream shall write the cache at the end of the file, plus the incomplete remaining bits. A "new" (not written and not dumped) cache is a single byte, and is guaranteed to be filled with zero `(\0 \0 \0 \0 \0 \0 \0 \0)`.

When dealing with reading from a stream, since Bel also reads bit by bit only, we take the same advantage of the cache, however as the opposite approach: we read a single byte from stream and keep the cache full. As we read each bit, we convert it to a Bel character (`\0` or `\1`). Once all bits of the cache have been read, another byte is fetched, stored on cache, and so on.

```
typedef enum BEL_STREAM_STATUS
{
    BEL_STREAM_CLOSED,
    BEL_STREAM_READ,
    BEL_STREAM_WRITE
} BEL_STREAM_STATUS;

typedef struct
{
    BEL_STREAM_STATUS  status;
    FILE              *raw_stream;
    uint8_t            cache;
    uint8_t            cache_used;
} Bel_stream;
```

4.6 Number

Bel does not specify any numeric types in its standard. In fact, numbers could be reproduced in Bel by using Church numerals, for example. However, this approach has a huge impact on performance, enough to make us want actual numeric types in our interpreter.

A *number* in Believe is a union of many number subtypes. The *number* can be an integer, a float, a fraction or even a complex number in its constitution, but this coercion happens away from the eyes of the Bel programmer; from his standpoint, there is only an opaque *number* type.

Let's start by defining the enumeration of types. Integers are C *64-bit signed ints*, and floats are, in fact, C *doubles*.

```
typedef enum {
    BEL_NUMBER_INT,
    BEL_NUMBER_FLOAT,
    BEL_NUMBER_FRACTION,
    BEL_NUMBER_COMPLEX
} BEL_NUMBER_TYPE;

typedef int64_t Bel_longint;
typedef double  Bel_float;
```

We forward declare the `Bel_number` structure as a *typedef* for a `struct BEL_NUMBER`.

```
typedef struct BEL_NUMBER Bel_number; // Forward declaration
```

Now we define our *fraction* and *complex* subtypes. Notice that they use `Bel_number` in their constitution. This is on purpose, as it allows us to create recursive definitions of numbers.

```
typedef struct {
    Bel *numer;
    Bel *denom;
} Bel_fraction;

typedef struct {
    Bel *real;
    Bel *imag;
} Bel_complex;
```

All that is left is to define our `Bel_number` formally.

```
struct BEL_NUMBER {
    BEL_NUMBER_TYPE type;
    union {
        Bel_longint  num_int;
        Bel_float    num_float;
        Bel_fraction num_frac;
        Bel_complex  num_compl;
    };
};
```

4.7 The `Bel` structure

The remaining thing to do is join all the types into the `Bel` type, which will serve as our generic way of dealing with things.

```
// Aliased as 'Bel' before
struct BEL
{
    BEL_TYPE type;
    union {
        Bel_sym     sym;
        Bel_pair   *pair;
        Bel_char    chr;
        Bel_stream  stream;
        Bel_number  number;
    };
};
```

CHAPTER 5

Essential structures and manipulation of data

5.1 Basic definitions

These definitions relate to essential symbols of the Bel global environment. They also encode the symbols' position on the global symbol table, to be defined later.

```
#define BEL_NIL   ((Bel_sym)0)
#define BEL_T     ((Bel_sym)1)
#define BEL_O     ((Bel_sym)2)
#define BEL_APPLY ((Bel_sym)3)
```

The following symbols are axioms which are global to the program. One is expected to use them instead of creating new symbols, though it is not strictly necessary.

```
Bel *bel_g_nil;
Bel *bel_g_t;
Bel *bel_g_o;
Bel *bel_g_apply;
```

These other variables are responsible for holding other axioms on the system. More on then will be specified later.

```
Bel *bel_g_chars;
Bel *bel_g_ins_sys;
Bel *bel_g_outs_sys;
Bel *bel_g_ins;
Bel *bel_g_outs;
```

```
6 Bel *bel_g_prim;
7 Bel *bel_g_clo;
```

We may also define temporary variables for the global, lexical and dynamic environments.

```
1 Bel *bel_g_scope;
2 Bel *bel_g_globe;
3 Bel *bel_g_dynae;
```

Forward declarations

We need to forward declare a few functions which will be useful for certain operations. For example, it is important that we make a forward declaration of `bel_mkerror`, since the primitives should depend on it; also, providing `bel_mkstring` ensures that the error format can be easily created, and so on.

```
1 /* Forward declarations */
2 Bel *bel_mkerror(Bel *format, Bel *vars);
3 Bel *bel_mkstring(const char*);
4 Bel *bel_mksymbol(const char*);
5 Bel *bel_car(Bel*);
6 Bel *bel_cdr(Bel*);
7 Bel *bel_mklist(int, ...);
```

5.2 Predicates

It is important to have a few predicates which will help us check for errors. These predicates do not check for argument nullability (e.g. unmanaged pointers), so use it wisely and only on initialized data!

`symbolp`

`bel_symbolp` tests whether the element is a symbol.

```
1 #define bel_symbolp(x) ((x)->type==BEL_SYMBOL)
```

`nilp`

`bel_nilp` tests whether the element is the *symbol* `nil`.

```
1 #define bel_nilp(x)                                    \
2     (bel_symbolp(x) && ((x)->sym==BEL_NIL))
```

pairp

`bel_pairp` tests whether the element is a *pair*.

```
1 #define bel_pairp(x) ((x)->type==BEL_PAIR)
```

atomp

`bel_atomp` tests whether an element is not a *pair* – that is, if it is not "divisible".

```
1 #define bel_atomp(x) (!bel_pairp(x))
```

charp

`bel_charp` tests whether the object is a character.

```
1 #define bel_charp(x)                    \
2     (((x)->type==BEL_CHAR))
```

streamp

`bel_streamp` tests whether the object is a stream.

```
1 #define bel_streamp(x)                  \
2     (((x)->type==BEL_STREAM))
```

numberp

`bel_numberp` determines whether `x` is a number or not. Notice that numbers are non-standard to Bel's definition.

```
1 #define bel_numberp(x)                  \
2     ((x)->type==BEL_NUMBER)
```

idp

`bel_idp` tests whether an object is identical to another. According to the Bel specification, identity is stricter than equality: there is only one of each symbol and character. Pairs and streams are compared by their references, so they are identical if and only if they reside in the same memory address.

This is the first predicate that is implemented as a proper C function, and it is used only internally; therefore, it outputs a C integer value for truth and falsity.

```
1 int bel_idp_nums(Bel *x, Bel *y); // Forward declaration
```

```
1  int
2  bel_idp(Bel *x, Bel *y)
3  {
4      if(bel_symbolp(x))
5          return (x->sym == y->sym);
6      else if(bel_charp(x))
7          return (x->chr == y->chr);
8      else if(bel_numberp(x)) {
9          // Non-standard
10         return bel_idp_nums(x, y);
11     }
12
13     // For pairs and streams, check for
14     // pointer aliasing
15     return (x == y);
16 }
```

Numbers are non-standard, so we develop our own identity test for them: if two *numbers* have the same *subtype* (integer, float, fraction, complex) and the same *value*, they are identical. In the case of numbers with components (fraction, complex) we recursively test for component identity instead of comparing *values* directly.

```
1  int
2  bel_idp_nums(Bel *x, Bel *y)
3  {
4      if(x->number.type == y->number.type) {
5          switch(x->number.type) {
6          case BEL_NUMBER_INT:
7              return (x->number.num_int
8                      == y->number.num_int);
9          case BEL_NUMBER_FLOAT:
10             return (x->number.num_float
11                     == y->number.num_float);
12         case BEL_NUMBER_FRACTION:
13             return
14                 (bel_idp_nums(
15                     x->number.num_frac.numer,
16                     y->number.num_frac.numer)
17                  && bel_idp_nums(
18                      x->number.num_frac.denom,
19                      y->number.num_frac.denom));
20         case BEL_NUMBER_COMPLEX:
21             return
22                 (bel_idp_nums(
```

```
23                      x->number.num_compl.real,
24                      y->number.num_compl.real)
25                  && bel_idp_nums(
26                      x->number.num_compl.imag,
27                      y->number.num_compl.imag));
28          };
29      }
30      return 0;
31  }
```

errorp

`bel_errorp` tests whether a specific object is a list in the format `(lit err . rest)`.

```
1   int
2   bel_errorp(Bel *x)
3   {
4       if(!bel_pairp(x)) return 0;
5       if(!bel_idp(bel_car(x), bel_mksymbol("lit")))
6           return 0;
7       Bel *cdr = bel_cdr(x);
8       if(!bel_idp(bel_car(cdr), bel_mksymbol("err")))
9           return 0;
10      return 1;
11  }
```

proper-list-p

A proper list is any list which ends in an appropriate `nil` symbol. So for example, `(1 2 3)` is a proper list, but `(1 2 3 . 4)` is not. Compare how these lists can be expressed by using dot notation:

- `(1 . (2 . (3 . nil)))`
- `(1 . (2 . (3 . 4)))`

An empty list is considered a proper list as well.

`bel_proper_list_p` checks whether a list is indeed a proper list. We do that by traversing the list, pair by pair. If the *cdr* is `nil`, it is proper; if it is a pair, it proceeds with the traversal. But if the *cdr* is anything else, then it is not a proper list.

```
1   int
2   bel_proper_list_p(Bel *x)
3   {
```

```
4       if(!bel_pairp(x) && !bel_nilp(x))
5           return 0;
6
7       Bel *itr = x;
8       while(!bel_nilp(itr)) {
9           if(!bel_pairp(itr))
10              return 0;
11          itr = bel_cdr(itr);
12      }
13
14      return 1;
15  }
```

stringp

An object is a string if and only if:

- it is a proper list;
- it contains characters only.

`bel_stringp` tests for this. However, this first implementation is a little naïve, since it performs a proper list check, which involves traversing an entire list, and then it traverses the list again, checking for characters in the *car*. This overhead can be reduced in the future.

```
1   int
2   bel_stringp(Bel *x)
3   {
4       if(!bel_proper_list_p(x)) {
5           return 0;
6       }
7
8       Bel *itr = x;
9       while(!bel_nilp(itr)) {
10          Bel *car = bel_car(itr);
11
12          if(!bel_charp(car))
13              return 0;
14
15          itr = bel_cdr(itr);
16      }
17
18      return 1;
19  }
```

literalp

bel_literalp takes a proper list and tells whether the list is a literal, that is, if the first element of the list is the symbol lit.

```
1  int
2  bel_literalp(Bel *x)
3  {
4      if(!bel_proper_list_p(x))
5          return 0;
6
7      return bel_idp(bel_car(x),
8                     bel_mksymbol("lit"));
9  }
```

primitivep

bel_primitivep takes a literal and tests whether it is a primitive, that is, if the second element of the list is the symbol prim.

```
1  int
2  bel_primitivep(Bel *x)
3  {
4      return bel_literalp(x)
5          && bel_idp(bel_car(bel_cdr(x)),
6                     bel_mksymbol("prim"));
7  }
```

closurep

bel_closurep takes a literal and tests whether it is a closure, that is, if the second element of the list is the symbol clo.

```
1  int
2  bel_closurep(Bel *x)
3  {
4      return bel_literalp(x)
5          && bel_idp(bel_car(bel_cdr(x)),
6                     bel_mksymbol("clo"));
7  }
```

quotep

bel_quotep takes a proper list and determines whether it is a quoted form.

```
int
bel_quotep(Bel *x)
{
    if(!bel_proper_list_p(x))
        return 0;

    return bel_idp(bel_car(x),
                   bel_mksymbol("quote"));
}
```

number-list-p

`bel_number_list_p` determines whether `x` is a proper list of numbers.

```
int
bel_number_list_p(Bel *x)
{
    if(!bel_proper_list_p(x)) {
        return 0;
    }

    Bel *itr = x;
    while(!bel_nilp(itr)) {
        Bel *car = bel_car(itr);

        if(!bel_numberp(car))
            return 0;

        itr = bel_cdr(itr);
    }

    return 1;
}
```

5.3 Symbol Table and Symbols

The symbol table is an array that grows as necessary, doubling in size, but never shrinks on the program's lifetime. Each element of the table is a `const` C string.

We begin by defining such structure and a global symbol table.

```
typedef struct {
    const char **tbl;
    uint64_t     n_syms;
```

```
4       uint64_t      size;
5   } _Bel_sym_table;
```

```
1   static _Bel_sym_table g_sym_table;
```

To initialize the symbol table, we give it an initial size of four, just enough to enclose Bel's four fundamental symbols: `nil`, `t`, `o` and `apply`. Notice that the order of these symbols relate to their predefined macros, so any failure here is unexpected.

```
1   void
2   bel_sym_table_init(void)
3   {
4       g_sym_table.n_syms = 4;
5       g_sym_table.size   = 4;
6       g_sym_table.tbl    =
7           GC_MALLOC(g_sym_table.size * sizeof(char*));
8
9       g_sym_table.tbl[BEL_NIL]   = "nil";
10      g_sym_table.tbl[BEL_T]     = "t";
11      g_sym_table.tbl[BEL_O]     = "o";
12      g_sym_table.tbl[BEL_APPLY] = "apply";
13  }
```

The lookup function `bel_sym_table_find` does a linear search for the presented literal on the symbol table. However, if it doesn't find the symbol, it implicitly calls `bel_sym_table_add`, which appends the symbol to the table.

This is obviously not a very wise approach as it opens up for some exploits on interning symbols, but should be enough as long as these symbols are only really interned on `lit` or `quote` scopes.

```
1   Bel_sym bel_sym_table_add(const char*); // Forward declaration
2
3   Bel_sym
4   bel_sym_table_find(const char *sym_literal)
5   {
6       uint64_t i;
7       for(i = 0; i < g_sym_table.n_syms; i++) {
8           if(!strcmp(sym_literal, g_sym_table.tbl[i])) {
9               return i;
10          }
11      }
12
13      return bel_sym_table_add(sym_literal);
14  }
```

```
15
16 Bel_sym
17 bel_sym_table_add(const char *sym_literal)
18 {
19     if(g_sym_table.n_syms == g_sym_table.size) {
20         uint64_t new_size = 2 * g_sym_table.size;
21         g_sym_table.tbl = GC_REALLOC(g_sym_table.tbl,
22                                      new_size * sizeof(char*));
23         g_sym_table.size = new_size;
24     }
25     g_sym_table.tbl[g_sym_table.n_syms++] = sym_literal;
26     return (g_sym_table.n_syms - 1);
27 }
```

Eventually we'll also need to take a symbol and find its character counterpart. Since the table is immutable, we can do that instantaneously by taking the character string at the symbol's position on the table. Notice that we do not check whether the given argument is a symbol, since it is also an internal function.

```
1 const char*
2 bel_sym_find_name(Bel *sym)
3 {
4     return g_sym_table.tbl[sym->sym];
5 }
```

Last but not least, we create a proper tool to build a symbol. Just give it your desired symbol as a string literal and the runtime takes care of the rest.

```
1 Bel*
2 bel_mksymbol(const char *str)
3 {
4     Bel *ret  = GC_MALLOC(sizeof (*ret));
5     ret->type = BEL_SYMBOL;
6     ret->sym  = bel_sym_table_find(str);
7     return ret;
8 }
```

5.4 Pairs

Pairs are the kernel of every Lisp, so we need tools to manipulate them.

We begin by specifying the function which builds pairs. Notice that the function itself takes two references to values, so pairs cannot exist without their *car* and *cdr*.

```
Bel*
bel_mkpair(Bel *car, Bel *cdr)
{
    Bel *ret  = GC_MALLOC(sizeof (*ret));
    ret->type = BEL_PAIR;
    ret->pair = GC_MALLOC(sizeof (Bel_pair));
    ret->pair->car = car;
    ret->pair->cdr = cdr;
    return ret;
}
```

Now we may easily extract information from pairs, using the *car* and *cdr* operations.

```
Bel*
bel_car(Bel *p)
{
    if(bel_nilp(p))
        return bel_g_nil;

    if(!bel_pairp(p)) {
        return bel_mkerror(
            bel_mkstring("Cannot extract the car of ~a."),
            bel_mkpair(p, bel_g_nil));
    }

    return p->pair->car;
}
```

```
Bel*
bel_cdr(Bel *p)
{
    if(bel_nilp(p))
        return bel_g_nil;

    if(!bel_pairp(p)) {
        return bel_mkerror(
            bel_mkstring("Cannot extract the cdr of ~a."),
            bel_mkpair(p, bel_g_nil));
    }

    return p->pair->cdr;
}
```

Let's also build a utility to return the size of a list. This is a `O(n)` operation which takes a well-formed list and iterates over it.

Note that **calculating the length of something that is not a *proper list* makes no sense and will crash this operation**. So before calling `bel_length`, it is probably a good idea to check for a valid proper list using `bel_proper_list_p` or a similar procedure.

```
1  uint64_t
2  bel_length(Bel *list)
3  {
4      Bel *itr = list;
5      uint64_t len = 0;
6      while(!bel_nilp(itr)) {
7          len++;
8          itr = bel_cdr(itr);
9      }
10     return len;
11 }
```

We can also create a variadic function which implements Bel string creation from a number of arguments passed to that string. This is useful when creating lists from C.

`bel_mklist` asks for a number of elements and a variadic list of `Bel*` objects. Then it attempts to create a single pair for each object, and place the object itself in the *car* of that pair.

When another object is added into the list, yet another pair is created and so on, and this pair is set as the *cdr* of the previous pair.

```
1  Bel*
2  bel_mklist(int n_elem, ...)
3  {
4      if(n_elem <= 0) return bel_g_nil;
5
6      va_list args;
7      va_start(args, n_elem);
8
9      Bel *list_start = NULL;
10     Bel *list = NULL;
11
12     int i;
13     for(i = 0; i < n_elem; i++) {
14         Bel *newp =
15             bel_mkpair(va_arg(args, Bel*),
16                        bel_g_nil);
17         if(!list) {
18             list = newp;
19             list_start = list;
```

```
20          } else {
21              list->pair->cdr = newp;
22              list = newp;
23          }
24      }
25
26      if(!list_start)
27          return bel_g_nil;
28
29      return list_start;
30  }
```

5.5 Characters and Strings

Let's begin by adding a small function to wrap a character in a Bel object.

```
1  Bel*
2  bel_mkchar(Bel_char c)
3  {
4      Bel *ret  = GC_MALLOC(sizeof *ret);
5      ret->type = BEL_CHAR;
6      ret->chr  = c;
7      return ret;
8  }
```

Characters have the size of one byte, so if we take a single list of 8 `\1` and `\0` characters, we should be able to generate a bitmask of the corresponding character in question.

```
1  Bel*
2  bel_char_from_binary(Bel *list)
3  {
4      if(!bel_pairp(list)) {
5          return bel_mkerror(
6              bel_mkstring("The binary representation of "
7                           "a character must be a string of "
8                           "characters \\0 and \\1."),
9              bel_g_nil);
10     }
11
12     if(!bel_proper_list_p(list)) {
13         return bel_mkerror(
14             bel_mkstring("The object ~a is not a proper "
```

```
15                          "list, and therefore not a list "
16                          "of characters \\0 and \\1."),
17              bel_mkpair(list, bel_g_nil));
18      }
19
20      size_t len = bel_length(list);
21
22      if(len != 8) {
23          return bel_mkerror(
24              bel_mkstring("The binary representation of "
25                          "a character must have exactly "
26                          "eight characters \\0 or \\1."),
27              bel_g_nil);
28      }
29
30      Bel_char mask = '\0';
31      size_t i;
32      Bel *current = list;
33
34      for(i = 0; i < len; i++) {
35          Bel *bitchar = bel_car(current);
36
37          if(!bel_charp(bitchar)) {
38              return bel_mkerror(
39                  bel_mkstring("The provided binary "
40                               "representation of a "
41                               "character does not contain "
42                               "only characters."),
43                  bel_g_nil);
44          }
45
46          if(bitchar->chr != '0' && bitchar->chr != '1') {
47              return bel_mkerror(
48                  bel_mkstring("The binary representation of "
49                               "a character must have exactly "
50                               "eight characters \\0 or \\1."),
51                  bel_g_nil);
52          }
53
54          if(bitchar->chr == '1') {
55              mask |= (1 << (7 - i));
56          }
57          current = bel_cdr(current);
```

```
58      }
59      return bel_mkchar(mask);
60  }
```

Strings on the Bel environment are nothing more than a list of characters, therefore we need a way to convert C strings to proper Bel lists.

```
1   Bel*
2   bel_mkstring(const char *str)
3   {
4       size_t len = strlen(str);
5
6       if(len == 0)
7           return bel_g_nil;
8
9       Bel **pairs = GC_MALLOC(len * sizeof (Bel));
10
11      // Create pairs where CAR is a character and CDR is nil
12      size_t i;
13      for(i = 0; i < len; i++) {
14          Bel *chr  = GC_MALLOC(sizeof *chr);
15          chr->type = BEL_CHAR;
16          chr->chr  = str[i];
17          pairs[i]  = bel_mkpair(chr, bel_g_nil);
18      }
19
20      // Link all pairs properly
21      for(i = 0; i < len - 1; i++) {
22          pairs[i]->pair->cdr = pairs[i + 1];
23      }
24
25      return pairs[0];
26  }
```

We also add a utility to take back a Bel string and turn it into a garbage-collected C string.

Note that the errors it can produce are instead dumped to the console and we return a null pointer; proper manipulation of this function is a responsibility of the programmer, since this is an internal function.

```
1   char*
2   bel_cstring(Bel *belstr)
3   {
4       if(!bel_pairp(belstr)) {
```

```
5           puts("INTERNAL ERROR on bel_cstring: "
6                "argument is not a pair");
7           return NULL;
8       }
9
10      if(!bel_stringp(belstr)) {
11          puts("INTERNAL ERROR on bel_cstring: "
12               "argument is not a string");
13          return NULL;
14      }
15
16      uint64_t len = bel_length(belstr);
17      if(len == 0) return NULL;
18
19      char *str    = GC_MALLOC_ATOMIC((len + 1) * sizeof (*str));
20
21      Bel *itr     = belstr;
22      size_t i     = 0;
23
24      while(!bel_nilp(itr)) {
25          str[i] = bel_car(itr)->chr;
26          itr    = bel_cdr(itr);
27          i++;
28      }
29      str[i] = '\0';
30      return str;
31  }
```

5.6 Streams

We start by creating tools to manipulate streams. First, we create a raw stream from a file.

```
1   Bel*
2   bel_mkstream(const char* name, BEL_STREAM_STATUS status)
3   {
4       Bel *ret           = GC_MALLOC(sizeof *ret);
5       ret->type          = BEL_STREAM;
6
7       if(status == BEL_STREAM_CLOSED) {
8           return bel_mkerror(
9               bel_mkstring("Cannot create a stream with "
10                           "CLOSED status."),
```

```
11              bel_g_nil);
12      }
13
14      if(!strncmp(name, "ins", 3)) {
15          ret->stream.raw_stream = stdin;
16      } else if(!strncmp(name, "outs", 4)) {
17          ret->stream.raw_stream = stdout;
18      } else {
19          ret->stream.raw_stream =
20              fopen(name,
21                    status == BEL_STREAM_READ ? "rb" : "wb");
22
23          if(!ret->stream.raw_stream) {
24              return bel_mkerror(
25                  bel_mkstring("Unable to open stream ~a."),
26                  bel_mkpair(
27                      bel_mkstring(name), bel_g_nil));
28          }
29      }
30
31      ret->stream.status     = status;
32      ret->stream.cache      = 0u;
33      ret->stream.cache_used = 0u;
34      return ret;
35  }
```

One important thing to have is a function which inputs a single bit in a file. We use the previously defined cache system for that; by filling the bits from left to right, we'll enable output as a single bit.

First we define the function which dumps and resets the cache of a specific stream when the cache is full; this should come in handy when closing the stream as well. After that, we do the actual bit writing. And of course, writing a bit returns `t` or `nil` for success and failure; this will most likely not be external to the Bel environment itself, since a failure in writing must signal an error. But that is not the job for this primitive.

```
1  Bel*
2  bel_stream_dump_cache(Bel_stream *stream)
3  {
4      if(!fwrite(&stream->cache, 1, 1, stream->raw_stream)) {
5          return bel_g_nil;
6      }
7      stream->cache_used = 0u;
8      stream->cache      = 0u;
9      return bel_g_t;
```

```
10  }
11
12  Bel*
13  bel_stream_write_bit(Bel_stream *stream, Bel_char bit)
14  {
15      if(bit != '0' || bit != '1') {
16          return bel_mkerror(
17              bel_mkstring("Written bit must be represented "
18                           "as a character 0 or 1"),
19              bel_g_nil);
20      }
21
22      if(stream->status != BEL_STREAM_WRITE) {
23          return bel_mkerror(
24              bel_mkstring("Write stream is not at WRITE "
25                           "state"),
26              bel_g_nil);
27      }
28
29      if(stream->cache_used >= 8) {
30          return bel_stream_dump_cache(stream);
31      } else {
32          if(bit == '1') {
33              stream->cache |= (1 << (7 - stream->cache_used));
34          }
35          stream->cache_used++;
36      }
37
38      return bel_mkchar(bit);
39  }
```

We can take advantage of the same variables to read single bits from a file, as described before too. Keep the cache full, read single bits as Bel characters, fill the cache when the read bits are exhausted.

```
1   Bel*
2   bel_stream_fill_cache(Bel_stream *stream)
3   {
4       if(!fread(&stream->cache, 1, 1, stream->raw_stream)) {
5           // Return nil on EOF
6           return bel_g_nil;
7       }
8       stream->cache_used = 8;
9       return bel_g_t;
```

```
10  }
11
12  Bel*
13  bel_stream_read_bit(Bel_stream *stream)
14  {
15      if(stream->status != BEL_STREAM_READ) {
16          return bel_mkerror(
17              bel_mkstring("Read stream is not at READ "
18                           "state"),
19              bel_g_nil);
20      }
21
22      Bel *ret;
23      if(stream->cache_used == 0) {
24          ret = bel_stream_fill_cache(stream);
25          if(bel_nilp(ret)) {
26              return bel_mksymbol("eof");
27          }
28      }
29
30      uint8_t mask = (1 << (stream->cache_used - 1));
31      ret = bel_mkchar(((mask & stream->cache) == mask)
32                       ? ((Bel_char)'1') : ((Bel_char)'0'));
33      stream->cache_used--;
34      return ret;
35  }
```

We'll also need a tool to close a certain stream. Here we're being a little more careful, since streams are managed more directly, by using the C API. And of course, if we're dealing with output, dump the stream cache before closing the file.

```
1   Bel*
2   bel_stream_close(Bel *obj)
3   {
4       if(obj->type != BEL_STREAM) {
5           return bel_mkerror(
6               bel_mkstring("Cannot close something that "
7                            "is not a stream."),
8               bel_g_nil);
9       }
10
11      if(obj->stream.status == BEL_STREAM_CLOSED) {
12          return bel_mkerror(
13              bel_mkstring("Cannot close a closed stream."),
```

```
14              bel_g_nil);
15      }
16
17      // Dump cache before closing
18      if(obj->stream.status == BEL_STREAM_WRITE) {
19          bel_stream_dump_cache(&obj->stream);
20      }
21
22      if(!fclose(obj->stream.raw_stream)) {
23          obj->stream.raw_stream = NULL;
24          obj->stream.status     = BEL_STREAM_CLOSED;
25          return bel_g_t;
26      }
27
28      return bel_mkerror(
29          bel_mkstring("Error closing stream: ~a."),
30          bel_mkpair(
31              bel_mkstring(strerror(errno)),
32              bel_g_nil));
33  }
```

The default input and output streams are enclosed in Bel objects here, however they relate to `stdin` and `stdout` respectively. To the system, by default they have `nil` value.

```
1  void
2  bel_init_streams(void)
3  {
4      bel_g_ins      = bel_g_nil;
5      bel_g_outs     = bel_g_nil;
6      bel_g_ins_sys  = bel_mkstream("ins",  BEL_STREAM_READ);
7      bel_g_outs_sys = bel_mkstream("outs", BEL_STREAM_WRITE);
8  }
```

Stream manipulation safety

Since streams are defined taking advantage of the C API for manipulating files, unfortunately these demand careful usage on Bel programs. When handling streams, it is absolutely necessary to close them. The Boehm GC does not have finalizers for C bindings, so unfortunately it is not possible for now to call a finalizer which automatically closes the stream when the stream object is garbage collected.

5.7 Numbers

As stated before, numbers are not described in Bel specification, however we're implementing it for minimal ease and performance for arithmetic manipulation.

We've built a resilient and recursive model for constituting numbers, so we begin by arranging tools to create them.

Number generation

Integers are pretty straightforward: we just allocate a proper space and store them.

```
1 Bel*
2 bel_mkinteger(int64_t num)
3 {
4     Bel *ret             = GC_MALLOC(sizeof (*ret));
5     ret->type            = BEL_NUMBER;
6     ret->number.type     = BEL_NUMBER_INT;
7     ret->number.num_int = num;
8     return ret;
9 }
```

The same goes for the *float* type (which is actually a C double).

```
1 Bel*
2 bel_mkfloat(double num)
3 {
4     Bel *ret             = GC_MALLOC(sizeof (*ret));
5     ret->type            = BEL_NUMBER;
6     ret->number.type     = BEL_NUMBER_FLOAT;
7     ret->number.num_float = num;
8     return ret;
9 }
```

A fraction has a layer of complexity, though. We take a numerator and a denominator as *numbers*, but we need to make sure they are numbers. Plus, even if they were, we need to make sure that the denominator *is not zero*. However, the only checks we perform here are related to the *numberness* of numerator and denominator.

```
1 Bel*
2 bel_mkfraction(Bel *numer, Bel *denom)
3 {
4     if(!bel_numberp(numer)) {
5         return bel_mkerror(
6             bel_mkstring("The object ~a is not "
```

```
7                           "a number."),
8               bel_mkpair(numer, bel_g_nil));
9       }
10
11      if(!bel_numberp(denom)) {
12          return bel_mkerror(
13              bel_mkstring("The object ~a is not "
14                           "a number."),
15              bel_mkpair(numer, bel_g_nil));
16      }
17
18      Bel *ret                    = GC_MALLOC(sizeof (*ret));
19      ret->type                   = BEL_NUMBER;
20      ret->number.type            = BEL_NUMBER_FRACTION;
21      ret->number.num_frac.numer = numer;
22      ret->number.num_frac.denom = denom;
23      return ret;
24  }
```

We follow the same principle for a complex number: *real* and *imaginary* parts need to be a number themselves.

```
1   Bel*
2   bel_mkcomplex(Bel *real, Bel *imag)
3   {
4       if(!bel_numberp(real)) {
5           return bel_mkerror(
6               bel_mkstring("The object ~a is not "
7                            "a number."),
8               bel_mkpair(real, bel_g_nil));
9       }
10
11      if(!bel_numberp(imag)) {
12          return bel_mkerror(
13              bel_mkstring("The object ~a is not "
14                           "a number."),
15              bel_mkpair(imag, bel_g_nil));
16      }
17
18      Bel *ret                    = GC_MALLOC(sizeof (*ret));
19      ret->type                   = BEL_NUMBER;
20      ret->number.type            = BEL_NUMBER_COMPLEX;
21      ret->number.num_compl.real = real;
22      ret->number.num_compl.imag = imag;
```

```
23      return ret;
24  }
```

Number arithmetic

The following operations always happen between two numbers. We make sure they are of compatible types to perform these operations, and then we return numbers of a proper subtype afterwards.

1. Forward declarations

```
1  /* Forward declarations */
2  Bel *bel_num_add(Bel *x, Bel *y);
3  Bel *bel_num_sub(Bel *x, Bel *y);
4  Bel *bel_num_mul(Bel *x, Bel *y);
5  Bel *bel_num_div(Bel *x, Bel *y);
```

2. Coercion

   Let's start with subtype coercion. Given a number and a number type flag, we coerce that number to a new number of that subtype. Returns a new number, and does not modify the old one.

   Coercing a float to a fraction uses a naïve approach: we multiply the number by `10` until it has no significant digits on the decimal part. We count the `i` multiplications we've made, and then we build a fraction where the numerator is a truncated, converted to integer result, and the denominator is exactly ten to the power of `i`.

```
1  Bel*
2  bel_num_coerce(Bel *number, BEL_NUMBER_TYPE type)
3  {
4      if(number->number.type == type)
5          return number;
6
7      switch(number->number.type) {
8      case BEL_NUMBER_INT:
9      {
10         switch(type) {
11         case BEL_NUMBER_FLOAT:
12             return bel_mkfloat(
13                 (double)number->number.num_int);
14         case BEL_NUMBER_FRACTION:
15             return bel_mkfraction(
16                 number,
17                 bel_mkinteger(1));
18         case BEL_NUMBER_COMPLEX:
```

```
19              return bel_mkcomplex(
20                  number,
21                  bel_mkinteger(0));
22          default: break;
23          };
24      }
25      break;
26      case BEL_NUMBER_FLOAT:
27      {
28          switch(type) {
29          case BEL_NUMBER_INT:
30              return bel_mkinteger(
31                  (int64_t)trunc(number->number.num_float));
32          case BEL_NUMBER_FRACTION:
33          {
34              double num  = number->number.num_float;
35              double trun = trunc(num);
36              int i = 0;
37              while(num != trun) {
38                  num *= 10.0;
39                  trun = trunc(num);
40                  i++;
41              }
42              return bel_mkfraction(
43                  bel_mkinteger((int64_t)num),
44                  bel_mkinteger((int64_t)pow(10, i)));
45          }
46          case BEL_NUMBER_COMPLEX:
47              return bel_mkcomplex(number,
48                                   bel_mkfloat(0.0));
49          default: break;
50          };
51      }
52      break;
53      case BEL_NUMBER_FRACTION:
54      {
55          switch(type) {
56          case BEL_NUMBER_INT:
57          {
58              Bel *float_res =
59                  bel_num_div(
60                      bel_num_coerce(
61                          number->number.num_frac.numer,
```

```
62                          BEL_NUMBER_FLOAT),
63                      bel_num_coerce(
64                          number->number.num_frac.denom,
65                          BEL_NUMBER_FLOAT));
66
67              return bel_mkinteger(
68                  (int64_t)trunc(
69                      float_res->number.num_float));
70          }
71          case BEL_NUMBER_FLOAT:
72              return bel_num_div(
73                  bel_num_coerce(
74                      number->number.num_frac.numer,
75                      BEL_NUMBER_FLOAT),
76                  bel_num_coerce(
77                      number->number.num_frac.denom,
78                      BEL_NUMBER_FLOAT));
79          case BEL_NUMBER_COMPLEX:
80              return bel_mkcomplex(number,
81                                   bel_mkinteger(0));
82          default: break;
83          };
84      }
85      break;
86      case BEL_NUMBER_COMPLEX:
87      {
88          switch(type) {
89          case BEL_NUMBER_INT:
90          {
91              Bel *coerced =
92                  bel_num_coerce(
93                      number->number.num_compl.real,
94                      BEL_NUMBER_FLOAT);
95
96              return bel_mkinteger(
97                  (int64_t)trunc(
98                      coerced->number.num_float));
99          }
100         case BEL_NUMBER_FLOAT:
101             return bel_num_coerce(
102                 number->number.num_compl.real,
103                 BEL_NUMBER_FLOAT);
104         case BEL_NUMBER_FRACTION:
```

```
105                 return bel_num_coerce(
106                     number->number.num_compl.real,
107                     BEL_NUMBER_FRACTION);
108             default: break;
109             };
110         }
111         break;
112         default: break;
113         };
114 
115         return number;
116 }
```

3. Force same type

   The following function takes two numbers, and makes sure they both have a subtype where both retain full information. Returns a pair containing both numbers.

```
1  Bel*
2  bel_num_mksametype(Bel *x, Bel *y)
3  {
4      switch(x->number.type) {
5      case BEL_NUMBER_INT:
6          switch(y->number.type) {
7          case BEL_NUMBER_INT:
8              // int -> int -> int
9              return bel_mkpair(x, y);
10         case BEL_NUMBER_FLOAT:
11             // int -> float -> float
12             return bel_mkpair(
13                 bel_num_coerce(x, BEL_NUMBER_FLOAT),
14                 y);
15         case BEL_NUMBER_FRACTION:
16             // int -> fraction -> fraction
17             return bel_mkpair(
18                 bel_num_coerce(x, BEL_NUMBER_FRACTION),
19                 y);
20         case BEL_NUMBER_COMPLEX:
21             // int -> complex -> complex
22             return bel_mkpair(
23                 bel_num_coerce(x, BEL_NUMBER_COMPLEX),
24                 y);
25         default: break;
26         }
```

```
27          break;
28      case BEL_NUMBER_FLOAT:
29          switch(y->number.type) {
30          case BEL_NUMBER_INT:
31              // float -> int -> float
32              // duplicate
33              return bel_num_mksametype(y, x);
34          case BEL_NUMBER_FLOAT:
35              // float -> float -> float
36              // same type
37              return bel_mkpair(x, y);
38          case BEL_NUMBER_FRACTION:
39              // float -> fraction -> fraction
40              return bel_mkpair(
41                  bel_num_coerce(x, BEL_NUMBER_FRACTION),
42                  y);
43          case BEL_NUMBER_COMPLEX:
44              // float -> complex -> complex
45              return bel_mkpair(
46                  bel_num_coerce(x, BEL_NUMBER_COMPLEX),
47                  y);
48              break;
49          default: break;
50          }
51          break;
52      case BEL_NUMBER_FRACTION:
53          switch(y->number.type) {
54          case BEL_NUMBER_INT:
55              // fraction -> int -> int
56              // duplicate
57              return bel_num_mksametype(y, x);
58          case BEL_NUMBER_FLOAT:
59              // fraction -> float -> fraction
60              // duplicate
61              return bel_num_mksametype(y, x);
62          case BEL_NUMBER_FRACTION:
63              // fraction -> fraction -> fraction
64              // same type
65              return bel_mkpair(x, y);
66          case BEL_NUMBER_COMPLEX:
67              // fraction -> complex -> complex
68              return bel_mkpair(
69                  bel_num_coerce(x, BEL_NUMBER_COMPLEX),
```

```
70                      y);
71                  break;
72              default: break;
73              }
74              break;
75          case BEL_NUMBER_COMPLEX:
76              switch(y->number.type) {
77              case BEL_NUMBER_INT:
78                  // complex -> int -> complex
79                  // duplicate
80                  return bel_num_mksametype(y, x);
81              case BEL_NUMBER_FLOAT:
82                  // complex -> float -> complex
83                  // duplicate
84                  return bel_num_mksametype(y, x);
85              case BEL_NUMBER_FRACTION:
86                  // complex -> fraction -> complex
87                  // duplicate
88                  return bel_num_mksametype(y, x);
89              case BEL_NUMBER_COMPLEX:
90                  // complex -> complex -> complex
91                  // same type
92                  return bel_mkpair(x, y);
93              default: break;
94              }
95              break;
96          default: break;
97          }
98
99          // Satisfy the compiler on event of no coercion
100         return bel_mkpair(x, y);
101 }
```

a) Helper macro for functions

The following macro does an inline conversion of `Bel` pointers to same number subtype. Only the locals `x` and `y` will be affected; the original pointed objects won't be modified.

```
1  #define BEL_NUM_SAMETYPE(x, y)                    \
2      {                                             \
3      Bel *p = bel_num_mksametype(x, y);            \
4      x = bel_car(p);                               \
5      y = bel_cdr(p);                               \
6      }
```

4. Checking for zero

   This function checks whether the argument is zero.

   Comparing directly for zero on a double is not a really good idea. We're doing a naïve approach here, but it is not completely guaranteed.

```
1  int
2  bel_num_zerop(Bel *x)
3  {
4      switch(x->number.type) {
5      case BEL_NUMBER_INT:
6          return (x->number.num_int == 0);
7      case BEL_NUMBER_FLOAT:
8          return (x->number.num_float == 0.0)
9              || (x->number.num_float == -0.0);
10     case BEL_NUMBER_FRACTION:
11         return bel_num_zerop(
12             x->number.num_frac.numer);
13     case BEL_NUMBER_COMPLEX:
14         return (bel_num_zerop(
15                     x->number.num_compl.real))
16             && (bel_num_zerop(
17                     x->number.num_compl.imag));
18     }
19
20     // This should not be reached...
21     return 0;
22 }
```

5. Addition

   The following function adds two arbitrary numbers.

```
1  Bel*
2  bel_num_add(Bel *x, Bel *y)
3  {
4      BEL_NUM_SAMETYPE(x, y);
5
6      switch(x->number.type) {
7      case BEL_NUMBER_INT:
8          return bel_mkinteger(
9              x->number.num_int + y->number.num_int);
10     case BEL_NUMBER_FLOAT:
11         return bel_mkfloat(
12             x->number.num_float + y->number.num_float);
```

```
13      case BEL_NUMBER_FRACTION:
14      {
15          Bel *new_numer_x =
16              bel_num_mul(x->number.num_frac.numer,
17                          y->number.num_frac.denom);
18          Bel *new_numer_y =
19              bel_num_mul(x->number.num_frac.denom,
20                          y->number.num_frac.numer);
21          Bel *new_denom =
22              bel_num_mul(x->number.num_frac.denom,
23                          y->number.num_frac.denom);
24
25          return bel_mkfraction(
26              bel_num_add(new_numer_x, new_numer_y),
27              new_denom);
28      }
29      case BEL_NUMBER_COMPLEX:
30          return bel_mkcomplex(
31              bel_num_add(x->number.num_compl.real,
32                          y->number.num_compl.real),
33              bel_num_add(x->number.num_compl.imag,
34                          y->number.num_compl.imag));
35      default: break;
36      };
37
38      return bel_mkerror(
39          bel_mkstring("Error while adding ~a and ~a."),
40          bel_mkpair(x, bel_mkpair(y, bel_g_nil)));
41  }
```

6. Subtraction

This function is identical to `bel_num_add`, however it subtracts two numbers.

```
1   Bel*
2   bel_num_sub(Bel *x, Bel *y)
3   {
4       BEL_NUM_SAMETYPE(x, y);
5
6       switch(x->number.type) {
7       case BEL_NUMBER_INT:
8           return bel_mkinteger(
9               x->number.num_int - y->number.num_int);
10      case BEL_NUMBER_FLOAT:
```

```
11          return bel_mkfloat(
12              x->number.num_float - y->number.num_float);
13      case BEL_NUMBER_FRACTION:
14      {
15          Bel *new_numer_x =
16              bel_num_mul(x->number.num_frac.numer,
17                          y->number.num_frac.denom);
18          Bel *new_numer_y =
19              bel_num_mul(x->number.num_frac.denom,
20                          y->number.num_frac.numer);
21          Bel *new_denom =
22              bel_num_mul(x->number.num_frac.denom,
23                          y->number.num_frac.denom);
24
25          return bel_mkfraction(
26              bel_num_sub(new_numer_x, new_numer_y),
27              new_denom);
28      }
29      case BEL_NUMBER_COMPLEX:
30          return bel_mkcomplex(
31              bel_num_sub(x->number.num_compl.real,
32                          y->number.num_compl.real),
33              bel_num_sub(x->number.num_compl.imag,
34                          y->number.num_compl.imag));
35      default: break;
36      };
37
38      return bel_mkerror(
39          bel_mkstring("Error while subtracting ~a "
40                       "and ~a."),
41          bel_mkpair(x, bel_mkpair(y, bel_g_nil)));
42  }
```

7. Multiplication

   This function multiplies two arbitrary numbers.

```
1  Bel*
2  bel_num_mul(Bel *x, Bel *y)
3  {
4      BEL_NUM_SAMETYPE(x, y);
5
6      switch(x->number.type) {
7      case BEL_NUMBER_INT:
```

```
8           return bel_mkinteger(
9               x->number.num_int * y->number.num_int);
10      case BEL_NUMBER_FLOAT:
11          return bel_mkfloat(
12              x->number.num_float * y->number.num_float);
13      case BEL_NUMBER_FRACTION:
14          return bel_mkfraction(
15              bel_num_mul(x->number.num_frac.numer,
16                          y->number.num_frac.numer),
17              bel_num_mul(x->number.num_frac.denom,
18                          y->number.num_frac.denom));
19      case BEL_NUMBER_COMPLEX:
20      {
21          Bel *real =
22              bel_num_sub(
23                  bel_num_mul(x->number.num_compl.real,
24                              y->number.num_compl.real),
25                  bel_num_mul(x->number.num_compl.imag,
26                              y->number.num_compl.imag));
27          Bel *imag =
28              bel_num_add(
29                  bel_num_mul(x->number.num_compl.real,
30                              y->number.num_compl.imag),
31                  bel_num_mul(x->number.num_compl.imag,
32                              y->number.num_compl.real));
33
34          return bel_mkcomplex(real, imag);
35      }
36      break;
37      default: break;
38      };
39
40      return bel_mkerror(
41          bel_mkstring("Error while multiplying "
42                       "~a and ~a."),
43          bel_mkpair(x, bel_mkpair(y, bel_g_nil)));
44  }
```

8. Division

   This function divides two arbitrary numbers. Notice that we check whether the second argument is zero.

```
1  Bel*
2  bel_num_div(Bel *x, Bel *y)
```

```
3  {
4      BEL_NUM_SAMETYPE(x, y);
5
6      if(bel_num_zerop(y)) {
7          return bel_mkerror(
8              bel_mkstring("Cannot divide by zero."),
9              bel_g_nil);
10     }
11
12     switch(x->number.type) {
13     case BEL_NUMBER_INT:
14         if(x->number.num_int % y->number.num_int) {
15             return bel_mkfraction(x, y);
16         } else {
17             return bel_mkinteger(
18                 x->number.num_int / y->number.num_int);
19         }
20     case BEL_NUMBER_FLOAT:
21         return bel_mkfloat(
22             x->number.num_float / y->number.num_float);
23     case BEL_NUMBER_FRACTION:
24         return bel_mkfraction(
25             bel_num_mul(x->number.num_frac.numer,
26                         y->number.num_frac.denom),
27             bel_num_mul(x->number.num_frac.denom,
28                         y->number.num_frac.numer));
29     case BEL_NUMBER_COMPLEX:
30     {
31         Bel *numer = bel_mkcomplex(
32             bel_num_add(
33                 bel_num_mul(x->number.num_compl.real,
34                             y->number.num_compl.real),
35                 bel_num_mul(x->number.num_compl.imag,
36                             y->number.num_compl.imag)),
37             bel_num_add(
38                 bel_num_mul(
39                     bel_mkinteger(-1),
40                     bel_num_mul(x->number.num_compl.real,
                        ↪  y->number.num_compl.imag)),
41                 bel_num_mul(x->number.num_compl.imag,
42                             y->number.num_compl.real)));
43
44         Bel *denom = bel_num_add(
```

```
45                bel_num_mul(y->number.num_compl.real,
46                            y->number.num_compl.real),
47                bel_num_mul(y->number.num_compl.imag,
48                            y->number.num_compl.imag));
49
50            return bel_mkfraction(numer, denom);
51        }
52        default: break;
53        }
54
55        return bel_mkerror(
56            bel_mkstring("Error while dividing "
57                         "~a and ~a."),
58            bel_mkpair(x, bel_mkpair(y, bel_g_nil)));
59    }
```

5.8 Errors

Bel does not have a formal specification on errors in primitives, other than saying that there might be an `err` function which throws an error in the system.

I will therefore specify that, in Believe, an error is a literal (much like closures and primitives) which obeys the pattern...

```
(lit err format . args)
```

...where `lit` is the expected symbol for something that evaluates to itself, `err` is the symbol which specifies that the object is an error, `format` is a Bel string which contains a format for the given arguments, and `args` is a list of arguments which should be parsed within the format.

For a first implementation, I intend to make the format specification follow loosely the conventions of the `format` macro in Common Lisp, having `~a` as the format for any object and `~%` as the format for a new line, for example.

Here's how it could look like:

```
> (err "Cannot use ~a on ~a.~%" '(1 2 3) square)
Error: Cannot use (1 2 3) on (lit clo nil (x) (* x x)).
```

However, since this is a detail which can be implemented in Bel itself, we'll just go ahead and say that there is a string format and a list of arguments.

```
1  Bel*
2  bel_mkerror(Bel *format, Bel *arglist)
3  {
4      return bel_mkpair(
5          bel_mksymbol("lit"),
```

```
6           bel_mkpair(
7               bel_mksymbol("err"),
8               bel_mkpair(format, arglist)));
9   }
```

CHAPTER 6

Axioms

To save memory, some of the following things will be globally defined.

6.1 Variables and constants

Define global symbols which can be used across the program. These symbols should be used repeatedly, and that's why they were already declared. See the `bel_init` function to refer to their initialization.

```
1  void
2  bel_init_ax_vars(void)
3  {
4      bel_g_nil   = bel_mksymbol("nil");
5      bel_g_t     = bel_mksymbol("t");
6      bel_g_o     = bel_mksymbol("o");
7      bel_g_apply = bel_mksymbol("apply");
8
9      bel_g_prim  = bel_mksymbol("prim");
10     bel_g_clo   = bel_mksymbol("clo");
11 }
```

`bel_g_prim` is not part of the axiom variables, but we'll define it here since we'll need this symbol for generating primitives later.

6.2 List of all characters

First, we build an auxiliary function which converts an 8-bit number into a string, where each character represents a bit.

```
char*
bel_conv_bits(uint8_t num)
{
    char *str = GC_MALLOC_ATOMIC(9 * sizeof(*str));

    uint8_t i;
    for(i = 0; i < 8; i++) {
        int is_bit_set = num & (1 << i);
        str[7 - i] = is_bit_set ? '1' : '0';
    }
    str[8] = '\0';

    return str;
}
```

We build a list of all characters so that the specification gets happy. It will be stored in the previously defined `bel_g_chars` global variable. This might seem unecessary in the future, though.

The list is supposed to be built out of pairs, therefore we start by creating 255 `Bel` instances, representing list nodes; every node is supposed to hold the pointer to a `Bel_pair`. These pairs will be linked to one another: the *cdr* of the first `Bel_pair` (again, contained inside a `Bel` instance) points to the second `Bel`; the *cdr* of the second `Bel_pair` (also contained on its `Bel` instance) points to the third `Bel`, and so on. The last *cdr* of the last `Bel_pair`, also enclosed on a `Bel` instance, contains the symbol `nil`.

Now, we discuss what should be held in the *car* of each of these pairs. And that would be other pairs, which will hold the actual information we desire. Each of these secondary pairs is comprised of a character at its *car*, and a Bel string representing the bits of the character as its *cdr*.

```
void
bel_init_ax_chars(void)
{
    // Create a vector of 255 list nodes
    Bel **list = GC_MALLOC(255 * sizeof(*list));

    size_t i;
    for(i = 0; i < 255; i++) {
        // Build a pair which holds the character information
        Bel *pair = bel_mkpair(bel_mkchar((Bel_char)i),
        ↪  bel_mkstring(bel_conv_bits(i)));
        // Assign the car of a node to the current pair,
        // set its cdr temporarily to nil
        list[i] = bel_mkpair(pair, bel_g_nil);
```

```
14      }
15
16      // Assign each pair cdr to the pair on the front.
17      // Last pair should have a nil cdr still.
18      for(i = 0; i < 254; i++) {
19          list[i]->pair->cdr = list[i + 1];
20      }
21
22      // Hold reference to first element only
23      bel_g_chars = list[0];
24  }
```

6.3 Environment

Any environment is nothing but a list of pairs, where each pair `(var . val)` represents the binding of a specific symbol `var` to the value `val`.

We begin by creating a function which pushes, non-destructively, a new pair to any environment. The result is the new environment.

```
1  Bel*
2  bel_env_push(Bel *env, Bel *var, Bel *val)
3  {
4      Bel *new_pair = bel_mkpair(var, val);
5      return bel_mkpair(new_pair, env);
6  }
```

Notice that this non-destructive approach is important, since a lexical enviroment is supposed to extend the enviroment it is called on – for example, the environment of a function called from top-level is a list where the first elements are lexical bindings, and (conceptually) the latter elements are bindings belonging to the global environment.

Now we register all our axioms to our global environment. This way, a lookup operation on the global scope will yield proper values.

First, we define a macro which uses `bel_env_push` to modify the `globe` environment variable. This macro just takes a `SYMSTR`, turns it into a symbol, and generates a new environment, which is then assigned to the global environment.

```
1  #define BEL_ENV_GLOBAL_PUSH(SYMSTR, VAL)            \
2      (bel_g_globe =                                  \
3       bel_env_push(bel_g_globe,                      \
4                    bel_mksymbol(SYMSTR), VAL))
```

Initializing the global environment involves pushing certain values to it. But the dynamic and lexical environments are initialized to `nil`.

```
void
bel_init_ax_env(void)
{
    bel_g_globe = bel_g_nil;
    bel_g_dynae = bel_g_nil;
    bel_g_scope = bel_g_nil; // TODO: is this really necessary?

    BEL_ENV_GLOBAL_PUSH("chars", bel_g_chars);
    BEL_ENV_GLOBAL_PUSH("ins",   bel_g_ins);
    BEL_ENV_GLOBAL_PUSH("outs",  bel_g_outs);
}
```

Then, we create a lookup function. This function traverses an environment in linear time, so it is not fast, but it does its job. A lookup process either returns the associated value or returns `nil`.

```
Bel*
bel_env_lookup(Bel *env, Bel *sym)
{
    if(bel_nilp(env)) {
        return bel_g_nil;
    }

    if(!bel_symbolp(sym)) {
        return bel_mkerror(
            bel_mkstring("Cannot perform lookup of ~a, "
                         "which is not a symbol."),
            bel_mkpair(sym, bel_g_nil));
    }

    Bel *itr = env;
    while(!bel_nilp(itr)) {
        Bel *p = bel_car(itr);
        if(bel_car(p)->type == BEL_SYMBOL
           && bel_car(p)->sym == sym->sym) {
            return bel_cdr(p);
        }

        itr = bel_cdr(itr);
    }
    return bel_g_nil;
}
```

We also implement a proper lookup function which takes a lexical environment and a symbol. The function traverses all environments in order (dynamic, lexical, global)

to find the associated value of the given symbol. If the symbol is not found, returns an error.

```
Bel*
bel_lookup(Bel *lenv, Bel *sym)
{
    Bel *value;

    // Dynamic scope lookup
    value = bel_env_lookup(bel_g_dynae, sym);
    if(!bel_nilp(value)) {
        return value;
    }

    // Lexical scope lookup
    value = bel_env_lookup(lenv, sym);
    if(!bel_nilp(value)) {
        return value;
    }

    // Global scope lookup
    value = bel_env_lookup(bel_g_globe, sym);
    if(bel_nilp(value)) {
        return bel_mkerror(
            bel_mkstring("The symbol ~a is unbound."),
            bel_mkpair(sym, bel_g_nil));
    }

    return value;
}
```

Another thing to do is enable assignment. We begin by creating a function which finds a specific symbol on a specific environment and replaces its value by the given one. On success, it returns the symbol; on failure, it returns `nil`. If the environment is empty, we also return `nil`. Oh, we also don't check if the given symbol is really a symbol, since this is an internal function.

```
Bel*
bel_env_replace_val(Bel *env, Bel *sym, Bel *new_val)
{
    if(bel_nilp(env)) {
        return bel_g_nil;
    }

```

```
8       Bel *itr = env;
9       while(!bel_nilp(itr)) {
10          Bel *p = bel_car(itr);
11          if(bel_idp(sym, bel_car(p))) {
12              p->pair->cdr = new_val;
13              return sym;
14          }
15          itr = bel_cdr(itr);
16      }
17      return bel_g_nil;
18  }
```

We also need a function which takes the reference to an environment and a symbol, and *unbinds* that symbol from the value in the environment. This can be achieved by simply iterating over the list and "unlinking" the relevant pair. We also don't perform all the checks on this internal function.

This function might modify the environment passed as reference by argument. We only return a non-nil answer (which is the same environment, but modified) if and only if the unbinding was successful.

```
1   Bel*
2   bel_env_unbind(Bel **env, Bel *sym)
3   {
4       if(bel_nilp(*env)) {
5           return bel_g_nil;
6       }
7
8       // If first element is a match, return
9       // cdr of environment
10      if(bel_idp(bel_car(bel_car(*env)), sym)) {
11          *env = bel_cdr(*env);
12          return bel_g_t;
13      }
14
15      // Iterate looking at the next element always.
16      // If next element is a match, set current cdr
17      // to cdr of next element
18      Bel *itr = *env;
19      while(!bel_nilp(bel_cdr(itr))) {
20          Bel *p = bel_car(bel_cdr(itr));
21          if(bel_idp(bel_car(p), sym)) {
22              itr->pair->cdr = p->pair->cdr;
23              return bel_g_t;
24          }
```

```
25
26          itr = bel_cdr(itr);
27      }
28
29      // On no substitution, return nil
30      return bel_g_nil;
31  }
```

The assignment operation itself respects the hierarchy of environments, to be described in the next subsection. We attempt to make an assignment on the three kinds of environment (lexical – given as argument –, dynamic and global). If the assignment fails in any of these, the symbol is bound to the given new value, on the *global* environment.

```
1   Bel*
2   bel_assign(Bel *lenv, Bel *sym, Bel *new_val)
3   {
4       Bel *ret;
5
6       // Dynamic assignment
7       ret = bel_env_replace_val(bel_g_dynae, sym, new_val);
8       if(!bel_nilp(ret)) return sym;
9
10      // Lexical assignment
11      ret = bel_env_replace_val(lenv, sym, new_val);
12      if(!bel_nilp(ret)) return sym;
13
14      // Global assignment
15      ret = bel_env_replace_val(bel_g_globe, sym, new_val);
16      if(!bel_nilp(ret)) return sym;
17
18      // When not assignment was made, we push a global value
19      bel_g_globe = bel_env_push(bel_g_globe, sym, new_val);
20      return sym;
21  }
```

We proceed by the same principle for the actual unbinding function: we respect the hierarchy of environments. Like `bel_env_unbind`, this function might modify the passed environment, and that is why we take a reference to it.

```
1   Bel*
2   bel_unbind(Bel **lenv, Bel *sym)
3   {
4       Bel *ans;
5
```

```
6       // Dynamic unbinding
7       ans = bel_env_unbind(&bel_g_dynae, sym);
8       if(!bel_nilp(ans)) {
9           return sym;
10      }
11
12      // Lexical unbinding
13      ans = bel_env_unbind(lenv, sym);
14      if(!bel_nilp(ans)) {
15          return sym;
16      }
17
18      // Global unbinding
19      ans = bel_env_unbind(&bel_g_globe, sym);
20      if(!bel_nilp(ans)) {
21          return sym;
22      }
23
24      // On no unbinding, return nil
25      return bel_g_nil;
26  }
```

Types and hierarchy of environments

There are three kinds of environments in Bel: Global, Lexical and Dynamic. The global environment (`bel_g_globe`, `globe`) contains symbols which are always visible from all scopes. This environment lives for the lifetime of the interpreter.

The lexical environment (`bel_g_scope`, `scope`) contains symbols which are visible only inside the current scope, and lives for a short period of time, linked to its scope. It is the environment captured by closures, and also the environment created when a closure is applied (as a specific symbol is bound to evaluate a closure's body).

The dynamic environment (`bel_g_dynae`) is like the global environment on its regards to access (symbols are visible to the whole application). However, the dynamic environment lives for a short period of time, linked to the scope it is used.

In Bel, any symbol lookup is performed by traversing the environments in the following order: *Dynamic*, *Lexical*, *Global*.

Environment extension and capturing

Being a sequential list of pairs, where the values are pushed to their top, environments (such as the lexical) can share symbols. For example, suppose the following closure called `orig-fun`.

```
(def orig-fun (x y)
```

```
  (join (new-fun x) y))
```

Suppose further that this closure is applied to the symbols `foo` and `bar`. They are then bound respectively to `x` and `y`. The closure's lexical environment during application would look like this:

```
((y . bar)  (x . foo))
```

Suppose also that the closure `new-fun` is defined like this:

```
(def new-fun (x)
  (id x 'foo))
```

When `new-fun` is applied inside `orig-fun`, it captures `orig-fun`'s lexical environment. Additionally, `new-fun` binds `foo` (associated with the original `x` symbol) to a new `x` symbol. So `new-fun`'s lexical environment looks like this:

```
((x . foo)  (y . bar)  (x . foo))
```

Since the environment stacks up definitions, a lookup process begins at top (here displayed as the leftmost pair) and finds the first binding of the requested symbol that it can find. So in `new-fun`, the value associated to the symbol `x` can only be the first pair represented above; however, after the evaluation of `new-fun`, back at `orig-fun`, the associated value of `x` would be the last pair.

Another interesting fact is that, if `new-fun` were to make a blind assignment to `y` after being called inside `orig-fun`, `y`'s associated value would be changed in `orig-fun`'s lexical environment, so the new value of `y` would be seen not only at `new-fun`; it would still be different when we returned to `orig-fun`.

If `new-fun` were called from outside `orig-fun` (more specifically, at top level), such assignment to `y` would create a new binding on the global environment, effectively creating a new global variable.

6.4 Literals

Although literals have already been seen on error implementation, but here we reuse the concept to generate literals that should exist on the global environment.

A *literal* is a list, where the first element is the symbol `lit`. Literals are described like persistent quotes, since evaluating a quoted form strips away the quoting. A *literal* is what should be used to describe things that evaluate to themselves.

Literals follow the form `(lit . rest)`, where `lit` is a symbol, and `rest` is a proper list of things that should be treated as a literal.

Primitives and functions are internally described as *literals*.

The first thing to do is create a tool for generating a literal; in general, what it does is create a pair, where the *car* is the symbol `lit`, and the *cdr* is anything that should be treated as a literal.

```
Bel*
bel_mkliteral(Bel *rest)
{
    if(!bel_proper_list_p(rest)) {
        return bel_mkerror(
            bel_mkstring("The object ~a is not a "
                         "proper list to be turned "
                         "into a literal."),
            bel_mkpair(rest, bel_g_nil));
    }

    return bel_mkpair(bel_mksymbol("lit"),
                      rest);
}
```

Primitives

As stated above, primitives are represented as literals, since they evaluate to themselves. We start by defining a tool to create a certain primitive; it should be noted that, since primitives are internal to the Bel implementation, this function does not check for errors.

A primitive has the form `(lit prim name)`, where `lit` and `prim` are constant symbols, and `name` is a symbol for the primitive name.

```
Bel*
bel_mkprim(Bel *sym)
{
    return bel_mkliteral(
        bel_mkpair(bel_g_prim,
                   bel_mkpair(sym, bel_g_nil)));
}
```

The next definition is a macro where, given an environment `env` and a C string literal `x`, it generates a primitive for `x` and pushes it to the enviroment `env`.

```
#define BEL_REGISTER_PRIM(env, x)                \
    {                                            \
    Bel *sym = bel_mksymbol(x);                  \
    env = bel_env_push(env, sym,                 \
                       bel_mkprim(sym));         \
    }
```

Then we create a function where, given an environment `env`, it registers all Bel primitives on it, creating a new environment which is returned. Notice that this new environment is in fact making use of the original one.

```
1  Bel*
2  bel_gen_primitives(Bel *env)
3  {
4      // Primitive functions
5      BEL_REGISTER_PRIM(env, "id");
6      BEL_REGISTER_PRIM(env, "join");
7      BEL_REGISTER_PRIM(env, "car");
8      BEL_REGISTER_PRIM(env, "cdr");
9      BEL_REGISTER_PRIM(env, "type");
10     BEL_REGISTER_PRIM(env, "xar");
11     BEL_REGISTER_PRIM(env, "xdr");
12     BEL_REGISTER_PRIM(env, "sym");
13     BEL_REGISTER_PRIM(env, "nom");
14     BEL_REGISTER_PRIM(env, "wrb");
15     BEL_REGISTER_PRIM(env, "rdb");
16     BEL_REGISTER_PRIM(env, "ops");
17     BEL_REGISTER_PRIM(env, "cls");
18     BEL_REGISTER_PRIM(env, "stat");
19     BEL_REGISTER_PRIM(env, "coin");
20     BEL_REGISTER_PRIM(env, "sys");
21
22     // Primitive operators
23     BEL_REGISTER_PRIM(env, "+");
24     BEL_REGISTER_PRIM(env, "-");
25     BEL_REGISTER_PRIM(env, "*");
26     BEL_REGISTER_PRIM(env, "/");
27     BEL_REGISTER_PRIM(env, "<");
28     BEL_REGISTER_PRIM(env, "<=");
29     BEL_REGISTER_PRIM(env, ">");
30     BEL_REGISTER_PRIM(env, ">=");
31     BEL_REGISTER_PRIM(env, "=");
32
33     // Other primitives
34     BEL_REGISTER_PRIM(env, "err");
35     BEL_REGISTER_PRIM(env, "gc");
36
37     return env;
38 }
```

The last step is to have a function which pushes these primitives automatically to the `globe` environment.

```
1  void
2  bel_init_ax_primitives()
```

```
3 {
4     bel_g_globe = bel_gen_primitives(bel_g_globe);
5 }
```

Closures

Creating a closure is very straightforward. We take an environment and a list. Such list must have two elements, where the first is a lambda list, and the second is the body of the function.

```
1 Bel*
2 bel_mkclosure(Bel *lenv, Bel *rest)
3 {
4     return bel_mkliteral(
5         bel_mkpair(bel_g_clo,
6                    bel_mkpair(lenv, rest)));
7 }
```

CHAPTER 7

Printing

The following functions are used to print a certain object on standard output.

7.1 Forward declarations

We forward declare the `bel_print` function since printing pairs calls it for the pairs' parts.

```
1 void bel_print(Bel*);
2 void bel_print_closure(Bel*);
3 void bel_print_primitive(Bel*);
```

7.2 Printing pairs

The first function is a specialization for printing pairs in general. This function should also handle the printing of lists gracefully.

A closure is also a pair (a literal, to be more precise), but it has its own printing function, which we invoke if needed.

Similarly, a primitive is a primitive function whose body and environment are not explicit to the Bel language, so we also add a way to print them.

```
1 void
2 bel_print_pair(Bel *obj)
3 {
4     if(bel_nilp(obj)) return;
5
6     Bel *itr = obj;
7
```

```
8       // Handle printing closures
9       // and primitives
10      if(bel_closurep(obj)) {
11          bel_print_closure(obj);
12          return;
13      } else if(bel_primitivep(obj)) {
14          bel_print_primitive(obj);
15          return;
16      }
17
18      putchar('(');
19      while(!bel_nilp(itr)) {
20          Bel *car = bel_car(itr);
21          Bel *cdr = bel_cdr(itr);
22
23          bel_print(car);
24
25          if(bel_nilp(cdr)) {
26              break;
27          } else if(cdr->type != BEL_PAIR) {
28              putchar(' ');
29              putchar('.');
30              putchar(' ');
31              bel_print(cdr);
32              break;
33          }
34          putchar(' ');
35          itr = cdr;
36      }
37      putchar(')');
38  }
```

Printing functions

We also add a general way for printing closures. This also turns the closure printing safer, since we do not print the associated environment.

```
1   void
2   bel_print_closure(Bel *obj)
3   {
4       Bel *lambda_list =
5           bel_car(bel_cdr(bel_cdr(bel_cdr(obj))));
6       printf("#<function (fn ");
7       if(bel_nilp(lambda_list)) {
```

```
8           printf("()");
9       } else bel_print_pair(lambda_list);
10      printf(") {%p}>", (void*)obj);
11  }
```

Another useful function to have is a way to print primitives, which are Bel functions that work like axioms, defined through C code.

```
1   void
2   bel_print_primitive(Bel *obj)
3   {
4       Bel *name = bel_car(bel_cdr(bel_cdr(obj)));
5       printf("#<function (prim ");
6       bel_print(name);
7       putchar(')');
8       putchar('>');
9   }
```

7.3 Printing strings

A string is a very specific type of list: it is a proper list comprised only of characters. However, this function is not supposed to test for the object's type; instead, it must be called when we are certain that the object in question is a string.

```
1   void
2   bel_print_string(Bel *obj)
3   {
4       putchar('\"');
5       Bel *itr = obj;
6       while(!bel_nilp(itr)) {
7           Bel_char c = bel_car(itr)->chr;
8
9           switch(c) {
10          case '\a': printf("\\bel"); break;
11          default:   putchar(c);      break;
12          }
13
14          itr = bel_cdr(itr);
15      }
16      putchar('\"');
17  }
```

7.4 Printing streams

Printing a stream involves printing something that cannot be read back in, so it can be considered merely aestethic. I made an option of either printing that it is closed, or printing its status along with the raw pointer.

```
1  void
2  bel_print_stream(Bel *obj)
3  {
4      printf("#<stream :status ");
5      if(obj->stream.status == BEL_STREAM_CLOSED) {
6          printf("closed>");
7      } else {
8          switch(obj->stream.status) {
9          case BEL_STREAM_READ:  printf("input ");  break;
10         case BEL_STREAM_WRITE: printf("output "); break;
11         default: printf("unknown ");              break;
12         }
13         printf("{0x%08lx}>", (uint64_t)obj->stream.raw_stream);
14     }
15 }
```

7.5 Printing numbers

We develop a function to print an arbitrary number. The function takes the number itself and a parameter which tells whether the sign should be explicit (the reason for that will be evident soon).

To print an *integer*, the only thing to do is to print a `long int`. We prepend it with a plus if the number is positive and the explicit sign flag is on.

To print a *float*, we print a `double` with reduced notation. If the number is round, we append `.0` to it. We also follow the same rule of *integers* when prepending the plus sign.

A *fraction* is a pair of two numbers. We just enclose them in a textual representation like `#(f number)`, where `number` is the numerator and the denominator separated by a slash. These two components can also be numbers of any kind, so we print them recursively, without forcing the plus sign.

A *complex* is also a pair of two numbers of any kind, where the first number is the *real* part and the second number is the *imaginary* part, which multiplies `i`. So we enclose it in a textual representation like `#(c number)`, where `number` is a complex number in the form `R+Ai`. In this form, `R` is the real part, printed as any Bel number; `A` is the imaginary part, but we force it to print its sign on screen, and then we prepend it with an `i`. To force `A`'s sign to appear, we call this function recursively, with the `force_sign` flag active.

```
1  void
2  bel_print_number(Bel *num, int force_sign)
3  {
4      switch(num->number.type) {
5      case BEL_NUMBER_INT:
6          if(force_sign && (num->number.num_int >= 0))
7              putchar('+');
8          printf("%ld", num->number.num_int);
9          break;
10     case BEL_NUMBER_FLOAT:
11         if(force_sign && (num->number.num_float >= 0.0))
12             putchar('+');
13         printf("%lg", num->number.num_float);
14         // Trailing .0 on round number
15         if(num->number.num_float
16            == trunc(num->number.num_float)) {
17             printf(".0");
18         }
19         break;
20     case BEL_NUMBER_FRACTION:
21         printf("#(f ");
22         bel_print_number(num->number.num_frac.numer, 0);
23         putchar('/');
24         bel_print_number(num->number.num_frac.denom, 0);
25         putchar(')');
26         break;
27     case BEL_NUMBER_COMPLEX:
28         printf("#(c ");
29         bel_print_number(num->number.num_frac.numer, 0);
30         bel_print_number(num->number.num_frac.denom, 1);
31         printf("i)");
32         break;
33     default:
34         printf("#<\?\?\?>");
35         break;
36     }
37 }
```

7.6 Generic printing

The next function handles the printing of any data type. Notice that it does not automatically print a newline character.

```
1  void
2  bel_print(Bel *obj)
3  {
4      switch(obj->type) {
5      case BEL_SYMBOL:
6          printf("%s", g_sym_table.tbl[obj->sym]);
7          break;
8      case BEL_PAIR:
9          if(!bel_stringp(obj)) {
10             bel_print_pair(obj);
11         } else {
12             bel_print_string(obj);
13         }
14         break;
15     case BEL_CHAR:
16         if(obj->chr == '\a')
17             printf("\\bel"); // There is no Bel without \bel
18         else printf("\\%c", obj->chr);
19         break;
20     case BEL_STREAM:
21         bel_print_stream(obj);
22         break;
23     case BEL_NUMBER:
24         bel_print_number(obj, 0);
25         break;
26     default:
27         printf("#<\?\?\?>"); // wat
28         break;
29     };
30 }
```

CHAPTER 8

Evaluator

The evaluator is the most crucial part of the Bel system. We follow the pattern of the *metacircular evaluator*: by having two functions, `eval` and `apply`, we make them call themselves mutually, equipping them with auxiliary functions and special forms to produce a working interpreter for a Lisp language.

8.1 Forward declarations

These declarations specify the most crucial functions of the interpreter. Forward declarations are important for the mutual calling part.

```
1 /* Forward declarations */
2 Bel *bel_eval(Bel *exp, Bel *lenv);
3 Bel *bel_apply(Bel *proc, Bel *args);
4 Bel *bel_evlist(Bel *elist, Bel *lenv);
5 Bel *bel_apply_primop(Bel *sym, Bel *args);
6 Bel *bel_bind(Bel *vars, Bel *vals, Bel *lenv);
```

The following forward declarations are related to *special forms* on the evaluator. These special forms are handled outside of the *eval* function to make it more succinct.

```
1 /* Forward declarations */
2 Bel *bel_special_if(Bel *exp, Bel *lenv);
3 Bel *bel_special_quote(Bel *exp, Bel *lenv);
4 Bel *bel_special_dyn(Bel *rest, Bel *lenv);
5 Bel *bel_special_set(Bel *clauses, Bel *lenv);
```

8.2 The *eval* function

`bel_eval` is the *evaluation* function. The objective is to take a particular expression, identify what it is (whether it is a special form or a simple function application), and dispatch it accordingly.

When a simple application is performed, we take a list and consider that the first element is the symbol that the function is bound to. So we evaluate every element of the list, including the function, and then we *apply* the closure (produced by evaluation of the function) to the rest of the evaluated elements, which will be passed as arguments.

It is also important to notice that the closure captures the lexical environment where it is evaluated.

```
1  Bel*
2  bel_eval(Bel *exp, Bel *lenv)
3  {
4  /* #ifdef BEL_DEBUG */
5  /*     printf("eval>  "); */
6  /*     bel_print(exp); */
7  /*     putchar(10); */
8  /* #endif */
9
10     // numbers eval to themselves
11     if(bel_numberp(exp))
12         return exp;
13
14     // symbol
15     if(bel_symbolp(exp)) {
16         // If one of axiom symbols, eval to itself
17         if(bel_idp(exp, bel_g_nil)
18            || bel_idp(exp, bel_g_t)
19            || bel_idp(exp, bel_g_o)
20            || bel_idp(exp, bel_g_apply))
21             return exp;
22         // else lookup on table
23         return bel_lookup(lenv, exp);
24     }
25
26     // quote
27     if(bel_quotep(exp))
28         return bel_special_quote(exp, lenv);
29
30     // lit
31     else if(bel_literalp(exp))
32         return exp; // eval to itself
```

```
33
34      // string
35      else if(bel_stringp(exp))
36          return exp; // eval to itself
37
38      // Special forms
39      else if(bel_proper_list_p(exp)) {
40          // fn: closure
41          if(bel_idp(bel_car(exp), bel_mksymbol("fn")))
42              return bel_mkclosure(lenv, bel_cdr(exp));
43
44          // if
45          if(bel_idp(bel_car(exp), bel_mksymbol("if")))
46              return bel_special_if(exp, lenv);
47
48          // TODO:
49          // apply
50          // where (not straightforward)
51
52          // dyn
53          if(bel_idp(bel_car(exp), bel_mksymbol("dyn")))
54              return bel_special_dyn(bel_cdr(exp), lenv);
55
56          // after
57
58          // set (global binding)
59          if(bel_idp(bel_car(exp), bel_mksymbol("set")))
60              return bel_special_set(bel_cdr(exp), lenv);
61
62          // ccc (call/cc)
63          // thread (does not share dynamic binding)
64
65          // otherwise it is the case of an application
66          return bel_apply(bel_eval(bel_car(exp), lenv),
67                           bel_evlist(bel_cdr(exp), lenv));
68      }
69
70      return bel_mkerror(
71          bel_mkstring("~a is not a proper list "
72                       "for the application of "
73                       "a function."),
74          bel_mkpair(exp, bel_g_nil));
75  }
```

8.3 The *apply* function

`bel_apply` is the *application* function. It takes a certain *function* and applies to the *list of evaluated arguments*. A function can be a primitive, but can also be a *literal closure*.

To apply a *closure*, we bind all arguments to the closure's formal parameters, creating an extended lexical environment; then we proceed to evaluate the closure's body under that new lexical environment.

```
1  Bel*
2  bel_apply(Bel *fun, Bel *args)
3  {
4  /* #ifdef BEL_DEBUG */
5  /*     printf("apply> "); */
6  /*     bel_print(fun); */
7  /*     printf(" -> "); */
8  /*     bel_print(args); */
9  /*     putchar(10); */
10 /* #endif */
11
12     // Check for errors on fun
13     if(bel_errorp(fun)) {
14         return fun;
15     }
16
17     // Primitive procedure
18     else if(bel_primitivep(fun)) {
19         return bel_apply_primop(
20             bel_car(bel_cdr(bel_cdr(fun))),
21             args);
22     }
23
24     // Closure
25     else if(bel_closurep(fun)) {
26         Bel *lenv =
27             bel_car(
28                 bel_cdr(bel_cdr(fun)));
29         Bel *lambda_list =
30             bel_car(
31                 bel_cdr(bel_cdr(bel_cdr(fun))));
32         Bel *body =
33             bel_car(
34                 bel_cdr(bel_cdr(bel_cdr(
35                                     bel_cdr(fun)))));
```

```
36
37          // Generate a new environment with the
38          // arguments bound in it
39          Bel *new_env = bel_bind(lambda_list,
40                                  args,
41                                  lenv);
42
43          if(bel_errorp(new_env)) {
44              return new_env;
45          }
46
47          // Evaluate body on the new environment
48          return bel_eval(body, new_env);
49      }
50
51      // Error
52      else {
53          return bel_mkerror(
54              bel_mkstring("~a is not a procedure"),
55              bel_mkpair(fun, bel_g_nil));
56      }
57  }
```

8.4 Auxiliary functions

The following functions are also essential to the evaluator, but have a more secondary role, such as handling special forms, applying primitive operators, and other kinds of things.

Evaluating special forms

Some special forms require greater attention, and so it is a little better to give them their own function.

1. `(quote x)`

   In Lisp languages, quoting an atom, like the expression `'a`, translates to an expression such as `(quote a)`, which will then be evaluated by returning only the symbol `a`.

```
1  Bel*
2  bel_special_quote(Bel *exp, Bel *lenv)
3  {
4      uint64_t len = bel_length(exp);
```

```
5       if(len != 2) {
6           return bel_mkerror(
7               bel_mkstring("Malformed quote: can only "
8                            "quote one object."),
9               bel_g_nil);
10      }
11
12      return bel_car(bel_cdr(exp));
13  }
```

2. `(if . clauses)`

   The conditional `if` takes any number of clauses (at least two), and does their evaluation in pairs of clauses (not to be confused with the pair data type).

   Suppose that we have a conditional such as

   ```
   (if cond1 pred1 cond2 pred2)
   ```

   We evaluate `cond1`. If its result is not `nil`, we return the evaluation of `pred1`.

   If evaluation of `cond1` is `nil`, however, we don't evaluate `pred1`; we proceed to test the evaluation of `cond2`. If `cond2` yields a non-`nil` result, however, we return the evaluation of `pred2`.

   There can also be a different scenario, where the number of clauses is odd, like

   ```
   (if cond1 pred1 cond2 pred2 altern)
   ```

   If, during evaluation, `cond2` did not yield a non-`nil` result, then `pred2` would be skipped; however, as there are no more pairs, but only the a single `altern` clause, it will be evaluated and its results will be returned, as an alternative.

```
1   Bel*
2   bel_special_if(Bel *exp, Bel *lenv)
3   {
4       Bel *body       = bel_cdr(exp);
5       uint64_t length = bel_length(body);
6
7       if(length < 2) {
8           return bel_mkerror(
9               bel_mkstring("if statement must have at "
10                           "least one predicate with "
11                           "a consequent."),
12              bel_g_nil);
13      }
```

```
14
15      Bel *predicate;
16      Bel *consequent;
17
18      while(1) {
19          predicate  = bel_car(body);
20          consequent = bel_car(bel_cdr(body));
21          body = bel_cdr(bel_cdr(body));
22
23          // nil consequent = return-eval predicate
24          if(bel_nilp(consequent)) {
25              return bel_eval(predicate, lenv);
26          }
27
28          if(!bel_nilp(bel_eval(predicate, lenv))) {
29              return bel_eval(consequent, lenv);
30          }
31      }
32
33      return bel_g_nil;
34  }
```

3. `(dyn v x y)`

   The special form `dyn` evaluates `x` and *dynamically* binds it to symbol `v`. After this dynamic binding, it then evaluates `y`, and finally unbinds `x`.

   This implementation of `dyn` is not thread-happy yet, since every thread is supposed to have its dynamic bindings, which are not shared. This, however, is something that will be solved later.

```
1   Bel*
2   bel_special_dyn(Bel *rest, Bel *lenv)
3   {
4       uint64_t len = bel_length(rest);
5
6       if(len > 3) {
7           return bel_mkerror(
8               bel_mkstring("Too many arguments on "
9                            "dynamic binding."),
10              bel_g_nil);
11      }
12
13      Bel *sym = bel_car(rest);
14      Bel *x   = bel_car(bel_cdr(rest));
```

```
15      Bel *y   = bel_car(bel_cdr(bel_cdr(rest)));
16
17      if(!bel_symbolp(sym)) {
18          return bel_mkerror(
19              bel_mkstring("Dynamic bindings can only "
20                           "be attributed to symbols."),
21              bel_g_nil);
22      }
23
24      if(bel_nilp(sym)) {
25          return bel_mkerror(
26              bel_mkstring("Cannot bind value to nil."),
27              bel_g_nil);
28      }
29
30  /* #ifdef BEL_DEBUG */
31  /*     printf("dynb>  "); */
32  /*     bel_print(sym); */
33  /*     printf(" := "); */
34  /*     bel_print(x); */
35  /*     putchar(10); */
36  /* #endif */
37
38      bel_g_dynae =
39          bel_env_push(bel_g_dynae,
40                       sym,
41                       bel_eval(x, lenv));
42
43      Bel *ret = bel_eval(y, lenv);
44      bel_env_unbind(&bel_g_dynae, sym);
45
46      return ret;
47  }
```

4. `(set . rest)`

   This form works with pairs in an expression like `(set s1 v1 s2 v2...)` in such a way that `vn` is evaluated and globally bound to `sn`.

   This is the form behind definitions of functions, for example.

   For the global assignments to happen properly, we evaluate all expressions before binding. This does not prevent side effects on the evaluation of values being assigned, but the values will only be assigned if no evaluation error happened. Plus, assignment can only be done to non `nil` symbols.

```
1  Bel*
2  bel_special_set(Bel *clauses, Bel *lenv)
3  {
4      Bel *syms = bel_g_nil;
5      Bel *vals = bel_g_nil;
6
7      Bel *itr = clauses;
8      while(!bel_nilp(itr)) {
9          Bel *sym = bel_car(itr);
10
11         if(!bel_symbolp(sym) || bel_nilp(sym)) {
12             return bel_mkerror(
13                 bel_mkstring("Global bindings can only "
14                              "be attributed to valid "
15                              "symbols."),
16                 bel_g_nil);
17         }
18
19         Bel *val =
20             bel_eval(bel_car(bel_cdr(itr)),
21                      lenv);
22
23         if(bel_errorp(val)) {
24             return val;
25         }
26
27         syms = bel_mkpair(sym, syms);
28         vals = bel_mkpair(val, vals);
29
30         itr = bel_cdr(bel_cdr(itr));
31     }
32
33     while(!bel_nilp(syms)) {
34 /* #ifdef BEL_DEBUG */
35 /*         printf("glob>  "); */
36 /*         bel_print(bel_car(syms)); */
37 /*         printf(" := "); */
38 /*         bel_print(bel_car(vals)); */
39 /*         putchar(10); */
40 /* #endif */
41         bel_assign(bel_g_nil, bel_car(syms),
           ↪  bel_car(vals));
42         syms = bel_cdr(syms);
```

```
43            vals = bel_cdr(vals);
44        }
45
46        return bel_g_nil;
47    }
```

Evaluate a list of values

`bel_evlist` evaluates a list of expressions under the given lexical environment. This function should only be called for a proper list.

```
1  Bel*
2  bel_evlist(Bel *elist, Bel *lenv)
3  {
4      if(bel_nilp(elist)) {
5          return bel_g_nil;
6      }
7
8      Bel *eval_result =
9          bel_eval(bel_car(elist), lenv);
10
11     if(bel_errorp(eval_result)) {
12         return eval_result;
13     }
14
15     Bel *ev_rest =
16         bel_evlist(bel_cdr(elist), lenv);
17
18     if(bel_errorp(ev_rest)) {
19         return ev_rest;
20     }
21
22     return bel_mkpair(eval_result, ev_rest);
23 }
```

Apply a primitive operator to a list

Applying a primitive to a list involves checking for the symbol which specifies it and dispatching the arguments to a specific function which checks arity and performs the job.

1. Forward declarations

   These forward declarations are related to the actual implementation of primitive functions in the Bel environment. We forward-declare them so that we can define

a function which redirects to each one of them, and after that we give their proper definitions.

```
1  /* Forward declarations of primitive functions */
2  Bel *bel_prim_id(Bel *args);
3  Bel *bel_prim_join(Bel *args);
4  Bel *bel_prim_car(Bel *args);
5  Bel *bel_prim_cdr(Bel *args);
6  Bel *bel_prim_type(Bel *args);
7  Bel *bel_prim_xar(Bel *args);
8  Bel *bel_prim_xdr(Bel *args);
9  Bel *bel_prim_sym(Bel *args);
10 Bel *bel_prim_nom(Bel *args);
11 Bel *bel_prim_wrb(Bel *args);
12 Bel *bel_prim_rdb(Bel *args);
13 Bel *bel_prim_ops(Bel *args);
14 Bel *bel_prim_cls(Bel *args);
15 Bel *bel_prim_stat(Bel *args);
16 Bel *bel_prim_coin(Bel *args);
17 Bel *bel_prim_sys(Bel *args);
18
19 /* Forward declarations of primitive operators */
20 Bel *bel_prim_add(Bel *args);
21 Bel *bel_prim_sub(Bel *args);
22 Bel *bel_prim_mul(Bel *args);
23 Bel *bel_prim_div(Bel *args);
24 //Bel *bel_prim_less(Bel *args);
25 //Bel *bel_prim_leq(Bel *args);
26 //Bel *bel_prim_great(Bel *args);
27 //Bel *bel_prim_geq(Bel *args);
28 //Bel *bel_prim_eq(Bel *args);
29
30 /* Forward declarations of other primitives */
31 Bel *bel_prim_err(Bel *args);
32 Bel *bel_prim_gc(Bel *args);
```

2. Applying primitive operations

   The `bel_apply_primop` function is the function which applies a primitive operation, identified as a symbol, to a list of evaluated values. It is important to know that this function does not usually do the job; instead, we just dispatch the arguments to a function which will perform as needed.

   The macro `bel_is_prim` compares whether `sym` is the symbol which represents the literal `lit`.

```
#define bel_is_prim(sym, lit)                          \
    (bel_idp(sym, bel_mksymbol(lit)))
```

The macro `bel_unimplemented` takes the symbol `sym` for a primitive function and generates an error, stating that the function has not been implemented. This is important while the interpreter is under development.

```
#define bel_unimplemented(sym)                         \
    bel_mkerror(                                       \
    bel_mkstring("~a is not implemented."),            \
    bel_mkpair(sym, bel_g_nil))
```

`bel_apply_primop` is the crucial function for the operations described above. It enumerates the core functions and dispatches the arguments accordingly.

Notice that an attempt to apply a primitive operation which does not exist results in error.

```
Bel*
bel_apply_primop(Bel *sym, Bel *args)
{
    // Primitive functions
    if(bel_is_prim(sym, "id"))
        return bel_prim_id(args);
    else if(bel_is_prim(sym, "join"))
        return bel_prim_join(args);
    else if(bel_is_prim(sym, "car"))
        return bel_prim_car(args);
    else if(bel_is_prim(sym, "cdr"))
        return bel_prim_cdr(args);
    else if(bel_is_prim(sym, "type"))
        return bel_prim_type(args);
    else if(bel_is_prim(sym, "xar"))
        return bel_prim_xar(args);
    else if(bel_is_prim(sym, "xdr"))
        return bel_prim_xdr(args);
    else if(bel_is_prim(sym, "sym"))
        return bel_prim_sym(args);
    else if(bel_is_prim(sym, "nom"))
        return bel_prim_nom(args);
    else if(bel_is_prim(sym, "wrb"))
        return bel_prim_wrb(args);
    else if(bel_is_prim(sym, "rdb"))
        return bel_prim_rdb(args);
    else if(bel_is_prim(sym, "ops"))
```

```
28          return bel_prim_ops(args);
29      else if(bel_is_prim(sym, "cls"))
30          return bel_prim_cls(args);
31      else if(bel_is_prim(sym, "stat"))
32          return bel_prim_stat(args);
33      else if(bel_is_prim(sym, "coin"))
34          return bel_prim_coin(args);
35      else if(bel_is_prim(sym, "sys"))
36          return bel_prim_sys(args);
37
38      // Primitive operators
39      else if(bel_is_prim(sym, "+"))
40          return bel_prim_add(args);
41      else if(bel_is_prim(sym, "-"))
42          return bel_prim_sub(args);
43      else if(bel_is_prim(sym, "*"))
44          return bel_prim_mul(args);
45      else if(bel_is_prim(sym, "/"))
46          return bel_prim_div(args);
47      else if(bel_is_prim(sym, "<"))
48          return bel_unimplemented(sym);
49      else if(bel_is_prim(sym, "<="))
50          return bel_unimplemented(sym);
51      else if(bel_is_prim(sym, ">"))
52          return bel_unimplemented(sym);
53      else if(bel_is_prim(sym, ">="))
54          return bel_unimplemented(sym);
55      else if(bel_is_prim(sym, "="))
56          return bel_unimplemented(sym);
57
58      // Other primitives
59      else if(bel_is_prim(sym, "err"))
60          return bel_prim_err(args);
61      else if(bel_is_prim(sym, "gc"))
62          return bel_prim_gc(args);
63
64      // Otherwise, unknown application operation
65      else {
66          return bel_mkerror(
67              bel_mkstring("Unknown primitive ~a."),
68              bel_mkpair(sym, bel_g_nil));
69      }
70  }
```

3. Maximum arity check

   The following macro is a helper for checking the arity of a specific function. Passing the arguments list and the number of desired arguments performs such a check. If the arity is greater than the given number, it returns an error complaining about it.

   Notice that this macro expects two things: to be called inside a function that returns a `Bel*` type, and that the arguments themselves are a *proper list*.

   It is also important to notice that Bel specifies that, when handling primitives, missing arguments default to `nil`, therefore passing less arguments than expected is not considered an error; since collecting arguments is handled by `bel_car` and `bel_cdr`, the missing arguments are guaranteed to be `nil` when retrieval is attempted.

```
#define BEL_CHECK_MAX_ARITY(args, num)                  \
    {                                                   \
    uint64_t length = bel_length(args);                 \
    if(length > num) {                                  \
    return bel_mkerror(                                 \
        bel_mkstring("Arity error"), bel_g_nil);        \
    }                                                   \
    }
```

4. Primitive functions

   The next functions implement primitive functions for the environment.

   a) `(id x y)`
   `id` checks whether `x` and `y` are identical. This is stricter than equality, since identity can only be tested for things that are always the same – namely, characters and symbols.

```
Bel*
bel_prim_id(Bel *args)
{
    BEL_CHECK_MAX_ARITY(args, 2);
    return (bel_idp(bel_car(args),
                    bel_car(bel_cdr(args)))
            ? bel_g_t : bel_g_nil);
}
```

   b) `(join x y)`
   `join` creates a pair with `x` as its *car* and `y` as its *cdr*.

```
Bel*
bel_prim_join(Bel *args)
{
```

```
4       BEL_CHECK_MAX_ARITY(args, 2);
5       return bel_mkpair(bel_car(args),
6                         bel_car(bel_cdr(args)));
7   }
```

c) `(car x)` and `(cdr x)`

`car` returns the first element of a pair `x`.

```
1   Bel*
2   bel_prim_car(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 1);
5       return bel_car(bel_car(args));
6   }
```

`cdr` returns the second element of a pair `x`.

```
1   Bel*
2   bel_prim_cdr(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 1);
5       return bel_cdr(bel_car(args));
6   }
```

d) `(type x)`

`type` returns a symbol which specifies the type of `x`. The returning values can be `symbol`, `pair`, `char`, `stream` or `number`.

```
1   Bel*
2   bel_prim_type(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 1);
5       switch(bel_car(args)->type) {
6       case BEL_SYMBOL:
7           return bel_mksymbol("symbol");
8           break;
9       case BEL_PAIR:
10          return bel_mksymbol("pair");
11          break;
12      case BEL_CHAR:
13          return bel_mksymbol("char");
14          break;
15      case BEL_STREAM:
16          return bel_mksymbol("stream");
17          break;
18      case BEL_NUMBER:
19          return bel_mksymbol("number");
```

```
20          break;
21      default:
22          return bel_mksymbol("unknown");
23          break;
24      };
25  }
```

e) `(xar x y)` and `(xdr x y)`

`xar` replaces the *car* of a pair `x` with the given value `y`.

```
1   Bel*
2   bel_prim_xar(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 2);
5       Bel *pair = bel_car(args);
6       Bel *val  = bel_car(bel_cdr(args));
7       if(!bel_pairp(pair)) {
8           return bel_mkerror(
9               bel_mkstring("~a is not a pair."),
10              bel_mkpair(pair, bel_g_nil));
11      }
12
13      pair->pair->car = val;
14      return val;
15  }
```

`xdr` replaces the *cdr* of a pair `x` with the given value `y`.

```
1   Bel*
2   bel_prim_xdr(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 2);
5       Bel *pair = bel_car(args);
6       Bel *val  = bel_car(bel_cdr(args));
7       if(!bel_pairp(pair)) {
8           return bel_mkerror(
9               bel_mkstring("~a is not a pair."),
10              bel_mkpair(pair, bel_g_nil));
11      }
12
13      pair->pair->cdr = val;
14      return val;
15  }
```

f) `(sym x)` and `(nom x)`

`sym` takes a Bel string and converts it into a symbol.

```
1  Bel*
2  bel_prim_sym(Bel *args)
3  {
4      BEL_CHECK_MAX_ARITY(args, 1);
5      Bel *str = bel_car(args);
6      if(!bel_stringp(str)) {
7          return bel_mkerror(
8              bel_mkstring("The object ~a must be a
                ↪  string."),
9              bel_mkpair(str, bel_g_nil));
10     }
11
12     char *cstr = bel_cstring(str);
13     if(!cstr || !strcmp(cstr, "")) {
14         return bel_mkerror(
15             bel_mkstring("The object ~a is not a proper
                ↪  string."),
16             bel_mkpair(str, bel_g_nil));
17     }
18
19     return bel_mksymbol(cstr);
20 }
```

`nom` takes a symbol and discovers its name as a Bel string.

```
1  Bel*
2  bel_prim_nom(Bel *args)
3  {
4      BEL_CHECK_MAX_ARITY(args, 1);
5      Bel *sym = bel_car(args);
6      if(!bel_symbolp(sym)) {
7          return bel_mkerror(
8              bel_mkstring("The object ~a is not a
                ↪  string."),
9              bel_mkpair(sym, bel_g_nil));
10     }
11
12     return bel_mkstring(
13         bel_sym_find_name(sym));
14 }
```

g) `(wrb x y)` and `(rdb x)`

`wrb` and `rdb` are functions responsible for input and output on a stream.

`wrb` takes a bit `x` and a stream `y`, and writes that bit to the stream. If the stream is `nil`, it writes instead to `outs`.

```
Bel*
bel_prim_wrb(Bel *args)
{
    BEL_CHECK_MAX_ARITY(args, 2);
    Bel *x = bel_car(args);
    Bel *y = bel_car(bel_cdr(args));

    if(!bel_charp(x)) {
        return bel_mkerror(
            bel_mkstring("The object ~a is not "
                         "a character."),
            bel_mkpair(x, bel_g_nil));
    }

    if(bel_nilp(y)) {
        y = bel_lookup(bel_g_nil,
        ↪  bel_mksymbol("outs"));
    } else {
        if(!bel_streamp(y)) {
            return bel_mkerror(
                bel_mkstring("The object ~a must be "
                             "a stream."),
                bel_mkpair(y, bel_g_nil));
        }
    }

    return bel_stream_write_bit(&y->stream, y->chr);
}
```

`rdb` simply reads a bit from the stream `x`.

```
Bel*
bel_prim_rdb(Bel *args)
{
    BEL_CHECK_MAX_ARITY(args, 1);
    Bel *x = bel_car(args);

    if(bel_nilp(x)) {
        bel_lookup(bel_g_nil, bel_mksymbol("ins"));
    } else {
        if(!bel_streamp(x)) {
            return bel_mkerror(
                bel_mkstring("The object ~a must be "
                             "a stream."),
                bel_mkpair(x, bel_g_nil));
```

```
15          }
16      }
17
18      return bel_stream_read_bit(&x->stream);
19  }
```

h) `(ops x y)`, `(cls x)` and `(stat x)`
`ops`, `cls` and `stat` are functions related to the status of a stream.
`ops` opens a stream to the file `x`, depending on the direction specified by symbol `y`, which can be either `in` or `out`.

```
1   Bel*
2   bel_prim_ops(Bel *args)
3   {
4       BEL_CHECK_MAX_ARITY(args, 2);
5       Bel *x = bel_car(args);
6       Bel *y = bel_car(bel_cdr(args));
7
8       if(!bel_stringp(x)) {
9           return bel_mkerror(
10              bel_mkstring("The object ~a is not "
11                           "a string."),
12              bel_mkpair(x, bel_g_nil));
13      }
14
15      if(!bel_symbolp(y)) {
16          return bel_mkerror(
17              bel_mkstring("The object ~a is not "
18                           "a symbol."),
19              bel_mkpair(y, bel_g_nil));
20      }
21
22      if(bel_idp(y, bel_mksymbol("in"))) {
23          return bel_mkstream(bel_cstring(x),
            ↪ BEL_STREAM_READ);
24      } else if(bel_idp(y, bel_mksymbol("out"))) {
25          return bel_mkstream(bel_cstring(x),
            ↪ BEL_STREAM_WRITE);
26      }
27
28      return bel_mkerror(
29          bel_mkstring("The object ~a is not one of "
30                       "the symbols `in` and `out`."),
31          bel_mkpair(y, bel_g_nil));
32  }
```

`cls` closes a stream `x`, as long as it is open. If it was closed, returns `t`; if it is already closed, returns `nil`.

```
Bel*
bel_prim_cls(Bel *args)
{
    BEL_CHECK_MAX_ARITY(args, 1);
    Bel *stream = bel_car(args);

    if(!bel_streamp(stream)) {
        return bel_mkerror(
            bel_mkstring("The object ~a is not "
                         "a stream."),
            bel_mkpair(stream, bel_g_nil));
    }

    if(stream->stream.status == BEL_STREAM_CLOSED) {
        return bel_g_nil;
    }

    return bel_stream_close(stream);
}
```

`stat` takes a stream `x` and gives back symbols `closed`, `in` or `out`, depending on stream status.

```
Bel*
bel_prim_stat(Bel *args)
{
    BEL_CHECK_MAX_ARITY(args, 1);
    Bel *stream = bel_car(args);
    if(!bel_streamp(stream)) {
        return bel_mkerror(
            bel_mkstring("The object ~a is not "
                         "a stream."),
            bel_mkpair(stream, bel_g_nil));
    }

    switch(stream->stream.status) {
    case BEL_STREAM_CLOSED: return
    ↪  bel_mksymbol("closed");
    case BEL_STREAM_READ:   return bel_mksymbol("in");
    case BEL_STREAM_WRITE:  return bel_mksymbol("out");
    default: // ...wat
        return bel_mkerror(
            bel_mkstring("The stream ~a has an "
```

```
20                              "unknown status."),
21              bel_mkpair(stream, bel_g_nil));
22      }
23  }
```

i) `(coin)`

`coin` returns symbols `t` and `nil` at random.

```
1  Bel*
2  bel_prim_coin(Bel *args)
3  {
4      BEL_CHECK_MAX_ARITY(args, 0);
5      return (rand() % 2) ? bel_g_t : bel_g_nil;
6  }
```

j) `(sys x)`

`sys` takes a string `x` and sends it to the operational system, as a console command, and returns the command's value as a proper Bel number.

Number types are non-standard to the Bel language, however Bel does not specify the return value of `sys`, therefore we have a degree of freedom to specify that the return of `sys` is a number.

This function specifically is somewhat a matter of concern, because it opens up for the execution of an arbitrary command on the operational system.

```
1  Bel*
2  bel_prim_sys(Bel *args)
3  {
4      BEL_CHECK_MAX_ARITY(args, 1);
5
6      Bel *str = bel_car(args);
7
8      if(!bel_stringp(str)) {
9          return bel_mkerror(
10             bel_mkstring("The object ~a is not "
11                          "a string."),
12             bel_mkpair(str, bel_g_nil));
13     }
14
15     const char *com = bel_cstring(str);
16
17     int64_t ret = system(com);
18
19     return bel_mkinteger(ret);
20 }
```

5. Primitive operators

These primitive operations take an arbitrary number of arguments and does the desired operation across the given values.

a) Addition

Adds all given values on the list, reducing them to a single number. If not argument is given, returns `0`. If called with a single argument, it returns that single argument (identity).

```
1  Bel*
2  bel_prim_add(Bel *args)
3  {
4      if(!bel_number_list_p(args)) {
5          return bel_mkerror(
6              bel_mkstring("Cannot add a non-number."),
7              bel_g_nil);
8      }
9
10     uint64_t length = bel_length(args);
11     // No args: return 0
12     if(length == 0) {
13         return bel_mkinteger(0);
14     }
15
16     // One arg: identity
17     if(length == 1) {
18         return bel_car(args);
19     }
20
21     Bel *ret = bel_car(args);
22     Bel *itr = bel_cdr(args);
23     while(!bel_nilp(itr)) {
24         ret = bel_num_add(ret, bel_car(itr));
25         itr = bel_cdr(itr);
26     }
27
28     return ret;
29 }
```

b) Subtraction

Subtracts all given values on the list, reducing them to a single number. If no argument is given, returns zero. If called with a single argument, inverts that argument, multiplying it by `-1`.

```
1  Bel*
2  bel_prim_sub(Bel *args)
```

```
{
    if(!bel_number_list_p(args)) {
        return bel_mkerror(
            bel_mkstring("Cannot subtract a "
                         "non-number."),
            bel_g_nil);
    }

    uint64_t length = bel_length(args);
    // No args: return zero
    if(length == 0) {
        return bel_mkinteger(0);
    }

    // One arg: invert
    if(length == 1) {
        return bel_num_mul(bel_mkinteger(-1),
                           bel_car(args));
    }

    Bel *ret = bel_car(args);
    Bel *itr = bel_cdr(args);
    while(!bel_nilp(itr)) {
        ret = bel_num_sub(ret, bel_car(itr));
        itr = bel_cdr(itr);
    }

    return ret;
}
```

c) Multiplication
Multiplies all given values on the list, reducing them to a single number. If no argument is given, returns `1`. If called with a single argument, returns that single argument (identity).

```
Bel*
bel_prim_mul(Bel *args)
{
    if(!bel_number_list_p(args)) {
        return bel_mkerror(
            bel_mkstring("Cannot multiply a "
                         "non-number."),
            bel_g_nil);
    }

```

```
11      uint64_t length = bel_length(args);
12      // No args: return 1
13      if(length == 0) {
14          return bel_mkinteger(1);
15      }
16
17      // One arg: identity
18      if(length == 1) {
19          return bel_car(args);
20      }
21
22      Bel *ret = bel_car(args);
23      Bel *itr = bel_cdr(args);
24      while(!bel_nilp(itr)) {
25          ret = bel_num_mul(ret, bel_car(itr));
26          itr = bel_cdr(itr);
27      }
28
29      return ret;
30  }
```

d) Division

Divides all given arguments on the given list, reducing them to a single number. If no argument is given, returns `1`. If called with a single argument, returns the given number.

If any division yields an error (e.g. a division by zero), returns that error immediately.

```
1   Bel*
2   bel_prim_div(Bel *args)
3   {
4       if(!bel_number_list_p(args)) {
5           return bel_mkerror(
6               bel_mkstring("Cannot divide a "
7                            "non-number."),
8               bel_g_nil);
9       }
10
11      uint64_t length = bel_length(args);
12      // No args: return 1
13      if(length == 0) {
14          return bel_mkinteger(1);
15      }
16
17      // One arg: return such number
```

```
18      if(length == 1) {
19          return bel_car(args);
20      }
21
22      Bel *ret = bel_car(args);
23      Bel *itr = bel_cdr(args);
24      while(!bel_nilp(itr)) {
25          ret = bel_num_div(ret, bel_car(itr));
26
27          // If there is a division by zero
28          // or something, return immediately
29          if(bel_errorp(ret)) {
30              return ret;
31          }
32
33          itr = bel_cdr(itr);
34      }
35
36      return ret;
37  }
```

6. Other primitives

These primitives are not specified in the Bel language, but are useful.

a) `(err x . rest)`

`err` creates an error using `x` as a format string, and appends the `rest` to the error as format arguments.

There is no arity check in `err`, though we do verify whether the first argument is a string. Problems with the arguments should appear when printing the error.

```
1   Bel*
2   bel_prim_err(Bel *args)
3   {
4       Bel *string = bel_car(args);
5       if(!bel_stringp(string)) {
6           return bel_mkerror(
7               bel_mkstring("First argument of `err` "
8                            "must be a string format."),
9               bel_g_nil);
10      }
11
12      // TODO: Maybe quote?
```

```
13      return bel_mkerror(string, bel_cdr(args));
14  }
```

b) `(gc)`
`gc` forces the garbage collector to perform garbage collection. Always returns `nil`.

```
1  Bel*
2  bel_prim_gc(Bel *args)
3  {
4      BEL_CHECK_MAX_ARITY(args, 0);
5      GC_gcollect();
6      return bel_g_nil;
7  }
```

Bind a list of variables to values

`bel_bind` binds each variable to an associated value. If the binding fails at any point, an error is returned; if not, a new environment with the bindings is returned.

```
1  Bel*
2  bel_bind(Bel *vars, Bel *vals, Bel *lenv)
3  {
4      int vars_ended = bel_nilp(vars);
5      int vals_ended = bel_nilp(vals);
6
7      if(vars_ended && !vals_ended) {
8          return bel_mkerror(
9              bel_mkstring("Too few variables in "
10                          "function application"),
11             bel_g_nil);
12     } else if(!vars_ended && vals_ended) {
13         return bel_mkerror(
14             bel_mkstring("Too few values in "
15                          "function application"),
16             bel_g_nil);
17     } else if(vars_ended && vals_ended) {
18         return lenv;
19     }
20
21     Bel *binding = bel_mkpair(bel_car(vars),
22                               bel_car(vals));
23
24     return bel_bind(bel_cdr(vars),
25                     bel_cdr(vals),
```

```
26                      bel_mkpair(binding, lenv));
27  }
```

CHAPTER 9

Reader

Our next job is to set up the reader. This section of our program is responsible for taking a specific amount of text and turning it into an actual program structure.

For example, an expression such as...

```
(set x (* 5 5))
```

...should have the same effect as the following block of C code:

```
1 bel_mklist(3,
2             bel_mksymbol("set"),
3             bel_mksymbol("x"),
4             bel_mklist(3,
5                        bel_mkinteger(5),
6                        bel_mkinteger(5)));
```

As it is true to every Lisp based on s-expressions, the parentheses determine a linked list of objects. It is a linked list, because it is a linking of pairs. In a more concrete way, we may express a list such as this:

```
(set . (x . ((* . (5 . (5 . nil))) . nil)))
```

Since the list is a concatenation of pairs, this also means that we can construct the whole thing by recursion and/or iteration on the usage of `bel_mkpair`, and leave `bel_mklist` as an internal function for when we want to write Bel lists from the C side of things. So the list could also, in principle, be constructed by this block of code too:

```
1 bel_mkpair(
2     bel_mksymbol("set"),
```

```
3       bel_mkpair(
4           bel_mksymbol("x"),
5           bel_mkpair(
6               bel_mkpair(
7                   bel_mksymbol("*"),
8                   bel_mkpair(
9                       bel_mkinteger(5),
10                      bel_mkpair(
11                          bel_mkinteger("5"),
12                          bel_g_nil))),
13              bel_g_nil)));
```

This is obviously very impractical, but it gives us a hint at how our parser should work. Every nested parentheses `()` indicates a pair whose `car` is also a pair; and for other elements, they should be interpreted as other data objects.

9.1 Tokenizer

The first step for parsing an expression is tokenization. This is where we will split our string into proper tokens (substrings), which will be stored as a proper Bel list of Bel strings.

This step also determines how to use *read macros*, which are not explicitly supported by Bel specification, but are useful for introducing new, different syntax. For example, a quoted list such as

```
'(1 2 3)
```

is, in fact, directly translated to

```
(quote (1 2 3))
```

In Believe, this makes it easier for us to parse these tokens later, because all we'll have to care about is recursively or iteratively build our lists of expressions, so any embellishment before that should be the a responsibility of the tokenization phase.

Believe has some inspiration on Lisp dialects such as Common Lisp, therefore implementing the tokenizer as a flexible stucture is a very good thing to do.

Basically, the main inspiration here is on Common Lisp's read macros. The idea is that one can reprogram the token reader to introduce custom syntax, not strictly bound to s-expressions.

The best part of having read macros is that some of the reading routines could be bootstrapped in Bel itself, so it opens the door for introducing languages built on top of Bel.

Reserved symbols

Some characters are very basic to Lisp, so we'll just go ahead and hardcode them. These symbols will automatically be excluded from token reading.

The following static vector has its end signalled by the null character.

```
static const char _Bel_reserved_chars[] = {
    '(', ')', '"', '[', ']', 0
};
```

An internal predicate will help us know if a certain character is reserved. It is designed to look similar to what you'll find at `ctypes.h`.

```
int
isreserved(char c)
{
    size_t i = 0;
    while(_Bel_reserved_chars[i] != 0) {
        if(c == _Bel_reserved_chars[i])
            return 1;
        i++;
    }
    return 0;
}
```

Read macros

The read macros themselves are exclusively programmed by using Bel, and they are an alist. Each element is another list with three elements at most.

```
((\')
 (\# \nul read-special-number))
```

As of now, read macros are still not implemented in Believe. In the future, this will be mixed with the reserved symbols and occupy the same space, managed by a single read table, which will be used by the rest of the tokenizer.

```
Bel *_Bel_rmacros;
```

```
Bel*
bel_init_read_macros()
{
    _Bel_rmacros =
        bel_mklist(1,
                   bel_mklist(3,
```

```
7                                   bel_mkchar('\''),
8                                   bel_g_nil,
9
                                    ↪ bel_mksymbol("read-special-quote")));
10      return bel_g_t;
11  }
```

```
1   int
2   isreadmacro(char c)
3   {
4       // TODO!
5       Bel *itr = _Bel_rmacros;
6       while(!bel_nilp(itr)) {
7           Bel *entry  = bel_car(itr);
8           Bel *charac = bel_car(entry);
9           if(charac->chr == c)
10              return 1;
11          itr = bel_cdr(itr);
12      }
13      return 0;
14  }
```

Token sizes

Before we begin the process of tokenization, it is important that we create a way to count the sizes of tokens so that we can separate the token into its own string.

The next function does that until it meets a reserved character or a space character.

```
1   size_t
2   token_length(const char *buffer, size_t position)
3   {
4       size_t i = position,
5           size = 0;
6       while(!isreserved(buffer[i]) &&
7             !isspace(buffer[i]) &&
8             (buffer[i] != '\0')) {
9           size++;
10          i++;
11      }
12      return size;
13  }
```

Let's also introduce a function for calculating token length of verbatim text. This is useful for text which should be read in verbatim, such as strings, for example. There, we

won't be looking for spaces or reserved symbols. We'll gobble up all characters, until we find a character which represents the end of text (and that character will be counted also).

However, if a null character is found, this means that the text was not properly finished. This is an error, so we'll return a size of `0` as a warning to the tokenizer.

```
1  size_t
2  token_verbatim_length(const char *buffer, size_t position, char
   ↪  end)
3  {
4      size_t i = position,
5          size = 0;
6      while(buffer[i] != end) {
7          if(buffer[i] == '\0')
8              return 0;
9          size++;
10         i++;
11     }
12     return size + 1;
13 }
```

Finally, we create a helper function which helps us copy a certain token into a proper Bel string.

```
1  Bel*
2  gen_tok_string(const char *buffer, size_t pos, size_t length)
3  {
4      char *ns = GC_MALLOC_ATOMIC((length + 1) * sizeof(char));
5      size_t i;
6      for(i = 0; i < length; i++) {
7          ns[i] = buffer[pos + i];
8      }
9      ns[length] = '\0';
10     return bel_mkstring(ns);
11 }
```

Tokenization

Tokenization is actually a simple process. We are going to write a procedure which creates a list and keeps appending Bel strings to it.

There seems to be quite a lot going on here, so here's an explanation, step by step.

Basically, this is a recursive function which builds a list of tokens. To make the operation faster, we also keep track of the last pair added (whose `car` is the last added token, and its `cdr` is always `nil`). Therefore, to add a new token, one must only create

a new pair in these molds with the relevant information, then replace the `cdr` of `last` by the address of this new pair.

This is a similar behavior to `xdr`, but we'll go ahead and do it manually.

As for the rest, it is just a case study. Characters are analyzed until we reach end of buffer. We perform tests in these orders:

1. Test if the character signals a read macro. Nothing is performed for now, since nothing falls into this case for now also.
2. Test if the character is `;`, signalling a comment. If so, it attempts to find the next line break. If the file ends with no line break, it is considered an error. The buffer pointer is then repositioned after the line break.
3. Test if the character is a double quote, signalling the beginning of a string literal. If so, it attempts to find the next double quote, and encloses both double quotes and the text inbetween to a new token. The buffer pointer is then repositioned after the second double quote.
4. Test if the character is reserved. If so, generates a token string with that same single character. The buffer pointer is then repositioned on the next character.
5. Test if the current character is not white space, indicating the beginning of a new arbitrary token. If so, it tries to find the next white space, then gobbles all characters before it into a new token. The buffer pointer is then repositioned at said white space.

```
1  Bel*
2  bel_tokenize(const char *buffer)
3  {
4      Bel *tokens = bel_g_nil;
5      Bel *last   = tokens;
6
7      size_t i;
8      for(i = 0; i < strlen(buffer); i++) {
9          Bel *token = bel_g_nil;
10         if(isreadmacro(buffer[i])) {
11             token = gen_tok_string(buffer, i, 1);
12         } else if(buffer[i] == ';') {
13             size_t length =
14                 token_verbatim_length(buffer, i, '\n');
15             if(length == 0) {
16                 return bel_mkerror(
17                     bel_mkstring("Unexpected EOF: "
18                                  "Comments must end on "
19                                  "line breaks"),
20                     bel_g_nil);
```

```
21              }
22              i += length;
23          } else if(buffer[i] == '"') {
24              size_t length =
25                  token_verbatim_length(buffer, i + 1, '"');
26              if(length == 0) {
27                  return bel_mkerror(
28                      bel_mkstring("Unbalanced double quote"),
29                      bel_g_nil);
30              }
31              token = gen_tok_string(buffer, i, length + 1);
32              i += length;
33          } else if(isreserved(buffer[i])) {
34              token = gen_tok_string(buffer, i, 1);
35          } else if(!isspace(buffer[i])) {
36              size_t length = token_length(buffer, i);
37              token = gen_tok_string(buffer, i, length);
38              i += length - 1;
39          } else {}
40
41          if(!bel_nilp(token)) {
42              if(bel_nilp(tokens)) {
43                  tokens = bel_mkpair(token, bel_g_nil);
44                  last   = tokens;
45              } else {
46                  last->pair->cdr = bel_mkpair(token, bel_g_nil);
47                  last = bel_cdr(last);
48              }
49          }
50      }
51      return tokens;
52  }
```

9.2 Parsing

Now we are going to work in the parser itself. Since the input is already tokenized, all we need to do is traverse the list of tokens, and build the list structure of our program.

For an expression such as

```
(set x (* 5 5))
```

we expect to have a proper Bel list of strings, which would be printed on console this way:

```
("(" "set" "x" "(" "*" "5" "5" ")" ")")
```

Every time we find an open parenthesis token `"("`, it indicates that we need to create a new list, which increases the depth of our parser's recursion, since it will be mainly comprised of a recursive function. The close parenthesis `")"` indicates that we need to close the previous list, unless we are at recursion depth `0`.

So this is basically a matter of recognizing the types of tokens and controlling the depth of recursion. We may also find some kinds of syntax errors here.

Forward declarations

Let's begin by declaring a few prototypes for important parsing functions. This time we'll be looking deeper into the tokens we split.

```
1 Bel *bel_parse_expr(Bel*, uint64_t);
2 Bel *bel_parse_token(Bel*);
3 Bel *bel_parse_int(const char*);
4 Bel *bel_parse_float(const char*);
5 Bel *bel_parse_frac(const char*);       // implement
6 Bel *bel_parse_char(const char*);       // implement
7 Bel *bel_parse_string(const char*);
```

We also need to declare prototypes for a few predicates, which are for the C part of the program. These will help us turn the tokens into proper symbols of their types.

```
1 int isstrint(const char*);
2 int isstrfloat(const char*);
3 int isstrfrac(const char*);    // implement
4 int isstrcomplex(const char*); // implement
5 int isstrchar(const char*);    // implement
6 int isstrstr(const char*);
```

Token list parser

Here, we are going to break a list of tokens into proper lists. At this point, everything should already be broken into proper s-expressions.

Parsing a token list is also a case analysis, much like we did on the tokenization phase. But this one is rather simple: we will take our string objects, convert them to C strings, then compare for parenthesis to perform recursion depth control.

When we find an open parenthesis, we recurse on the function. The expected result for `depth > 0` is a pair comprised of the parsed sublist of Bel symbols, and the yet-to-be-parsed rest of the list of tokens.

When we find a close parenthesis, either we build a pair with the current expression and the rest of tokens, or we generate an error if we're working on depth `0`.

As for all the other tokens, the token parsing job is dispatched to `bel_parse_token`, and the subexpression is appended to the results. If the subexpression is an error, though, we return it immediately.

```
1  Bel*
2  bel_parse_expr(Bel *tokens, uint64_t depth)
3  {
4      Bel *expr = bel_g_nil;
5      Bel *last = bel_g_nil;
6      /* Move next code to its own function!!! */
7      /* We can deal with read macros by parsing */
8      /* the next expression with it. */
9      while(!bel_nilp(tokens)) {
10         Bel *car           = bel_car(tokens);
11         tokens             = bel_cdr(tokens);
12         Bel *subexpr       = bel_g_nil;
13         const char *carstr = bel_cstring(car);
14
15         /* if((strlen(carstr) == 1) && isreadmacro(carstr[0]))
           ↪  { */
16         /*     // 1. Read the next expression */
17         /* } else  */if(!strcmp(carstr, ")")) {
18             if(depth == 0) {
19                 return bel_mkerror(
20                     bel_mkstring("Unbalanced parentheses"),
21                     bel_g_nil);
22             } else {
23                 return bel_mkpair(expr, tokens);
24             }
25         } else if(!strcmp(carstr, "(")) {
26             Bel *pair = bel_parse_expr(tokens, depth + 1);
27             subexpr   = bel_car(pair);
28             tokens    = bel_cdr(pair);
29         } else {
30             subexpr = bel_parse_token(car);
31         }
32
33         if(bel_errorp(subexpr)) {
34             return subexpr; // Errors out
35         }
36
37         if(bel_nilp(expr)) {
38             expr = bel_mkpair(subexpr, bel_g_nil);
39             last = expr;
```

```
40          } else {
41              last->pair->cdr =
42                  bel_mkpair(subexpr, bel_g_nil);
43              last = bel_cdr(last);
44          }
45      }
46      return expr;
47  }
```

Token parser

Parsing tokens themselves is a simple case analysis too. Here, we're taking the tokens as C strings and performing direct text comparisons on them.

Predicates such as `isstrnum`, `isstrfloat` and `isstrstr` help detect the type of information represented by the token. We then dispatch the text to functions of format `bel_parse_*`, which will then convert the token to a proper Bel object of the given kind.

```
1   Bel*
2   bel_parse_token(Bel *token)
3   {
4       const char *str = bel_cstring(token);
5       if(isstrint(str)) {
6           return bel_parse_int(str);
7       } else if(isstrfloat(str)) {
8           return bel_parse_float(str);
9       } else if(isstrstr(str)) {
10          return bel_parse_string(str);
11      }
12      // TODO: Add more special cases
13      return bel_mksymbol(str);
14  }
```

1. Parsing an integer

   We know that a token represents an integer if it is comprised of digits only. It may be preceeded by a minus sign too.

```
1   int
2   isstrint(const char *str)
3   {
4       uint64_t i = 0;
5
6       if(str[0] == '-') {
7           i++;
```

```
8       }
9
10      while(str[i] != '\0') {
11          if(!isdigit(str[i]))
12              return 0;
13          i++;
14      }
15      return 1;
16  }
```

If the token represents a proper integer, then we convert it to an `int64_t` by using `strtoll`, then we create the proper Bel number.

```
1   Bel*
2   bel_parse_int(const char *token)
3   {
4       return bel_mkinteger(strtoll(token, NULL, 10));
5   }
```

2. Parsing a float value

   A floating point may be syntactically preceeded by a minus, and *must* have a single dot anywhere.

```
1   int
2   isstrfloat(const char *str)
3   {
4       uint64_t i = 0;
5       int found_dot = 0;
6       if(str[0] == '-') {
7           i++;
8       } else if(str[0] == '.') {
9           found_dot = 1;
10          i++;
11      }
12
13      while(str[i] != '\0') {
14          if(str[i] == '.') {
15              if(!found_dot) {
16                  found_dot = 1;
17              } else {
18                  return 0;
19              }
20          } else if(!isdigit(str[i])) {
21              return 0;
```

```
22          }
23          i++;
24      }
25      return found_dot;
26  }
```

If the token is a proper Bel float, we use `strtod` to turn it into a `double` value of C language, then create the Bel object.

```
1  Bel*
2  bel_parse_float(const char *token)
3  {
4      return bel_mkfloat(strtod(token, NULL));
5  }
```

3. Parsing strings

   Anything between a pair of `"` is supposed to be a string, at least initially. So what we need to do is check for that.

```
1  int
2  isstrstr(const char *token)
3  {
4      return (token[0] == '"') &&
5          (token[strlen(token) - 1] == '"');
6  }
```

Having done so, all we need to do is create a new C string, properly trimming the token of its surrounding double quotes, then perform a proper conversion into a Bel string. Notice that the string size is calculated taking a null terminator into account.

```
1  Bel*
2  bel_parse_string(const char *token)
3  {
4      size_t strsz = strlen(token);
5      char *str = GC_MALLOC_ATOMIC((strsz - 1) *
       ↪  sizeof(char));
6      size_t i;
7      strsz--;
8      for(i = 0; i < strsz; i++) {
9          str[i] = token[i + 1];
10     }
11     str[strsz - 1] = '\0';
12     return bel_mkstring(str);
13 }
```

CHAPTER 10

REPL

We will now deal with the bits of our interpreter which are responsible for the user experience, namely the act of inputting a string which shall be evaluated by the program.

This is what we call the `REPL`. This is an acronym for the words "Read", "Evaluate", "Print", "Loop", which as indicated, describe precisely what it does: it is mostly comprised of a function, responsible for four high-level abstraction operations.

First, the function reads an input from console. This input is an arbitrary thing written by the user, and is just a string of characters. Then, this string is parsed into our proper internal structure of a list, where each element can be thought of as a valid form.

This generated list is then passed to the evaluator, which evaluates that string on an empty lexical environment. Upon evaluation, it generates a Bel object as response for each of the input forms on the given list, since it may contain multiple forms typed in a row, for example.

This new list of results is then printed to the console. Each of the results on the list is then printed on a different line.

Finally, we go back to the beginning, reading the input once again. This is what the *loop* word refers to.

10.1 Reading

10.2 Evaluation

10.3 Printing

10.4 Loop

CHAPTER 11

Debug

The following definitions are related to testing what we have so far.

11.1 Tests

String manipulation and printing

A string test which shows the conversion between C strings and Bel strings, and vice-versa.

```
1  void
2  string_test()
3  {
4      Bel *bel  = bel_mkstring("Hello, Bel!");
5      bel_print(bel);
6      printf(" => %s\n", bel_cstring(bel));
7
8      bel = bel_mkstring("There is no Bel without \a");
9      bel_print(bel);
10     putchar(10);
11 }
```

List/pair/dotted list notation

The following notation tests the printing capabilities of the list printing algorithm. Should be able to handle printing lists and dot-notation when necessary.

The data input reads as `((foo . bar) . (baz . quux))`, but the expected output is `((foo . bar) baz . quux)`.

```
1  void
2  notation_test()
3  {
4      Bel*
5      bel = bel_mkpair(bel_mkpair(bel_mksymbol("foo"),
6                                  bel_mksymbol("bar")),
7                       bel_mkpair(bel_mksymbol("baz"),
8                                  bel_mksymbol("quux")));
9      bel_print(bel);
10     putchar(10);
11 }
```

Proper list notation

This next test outputs the list `(The quick brown fox jumps over the lazy dog)`, which is a proper list of symbols.

```
1  void
2  list_test()
3  {
4      Bel*
5      bel =
6          bel_mklist(9,
7                     bel_mksymbol("The"),
8                     bel_mksymbol("quick"),
9                     bel_mksymbol("brown"),
10                    bel_mksymbol("fox"),
11                    bel_mksymbol("jumps"),
12                    bel_mksymbol("over"),
13                    bel_mksymbol("the"),
14                    bel_mksymbol("lazy"),
15                    bel_mksymbol("dog"));
16     bel_print(bel);
17     putchar(10);
18 }
```

Closure representation

This test is also a list of symbols, but with nested lists also. Plus, this is a proper list, representing the internal representation of a closure such as `(fn (x) (* x x))`, but in its expected output form, which is `(lit clo nil (x) (* x x))`. Plus, we try to represent another closure in its original syntax as `(fn (x) (+ 1 x))`.

We also take the oportunity to test `bel_mklist` by creating a proper closure not in its literal form.

```
1  void
2  closure_repr_test()
3  {
4      // (lit clo nil (x) (* x x))
5      Bel*
6      bel = bel_mklist(5,
7                       bel_mksymbol("lit"),
8                       bel_mksymbol("clo"),
9                       bel_g_nil,
10                      bel_mklist(1,
11                                 bel_mksymbol("x")),
12                      bel_mklist(3,
13                                 bel_mksymbol("*"),
14                                 bel_mksymbol("x"),
15                                 bel_mksymbol("x")));
16     bel_print(bel);
17     putchar(10);
18
19     // (fn (x) (+ 1 x))
20     bel =
21         bel_mklist(3, bel_mksymbol("fn"),
22                    bel_mklist(1, bel_mksymbol("x")),
23                    bel_mklist(3, bel_mksymbol("+"),
24                               bel_mkinteger(1),
25                               bel_mksymbol("x")));
26     bel_print(bel);
27     putchar(10);
28 }
```

Character list printing and environment lookup

This next test prints the first ten characters in the global `chars`, which is a list of pairs, each pair `(c . d)` containing a character `c`, and its string representation in binary `d`.

It is also interesting to notice that the `chars` global is obtained by a lookup operation on the environment, rather than using the global variable directly.

```
1  void
2  character_list_test()
3  {
4      // Character list
5      // Char: 000 (?) => "00000000"
6      // Char: 001 (?) => "00000001"
7      // etc
```

```
8       const int first_char = 'a';
9
10      Bel *bel = bel_env_lookup(bel_g_globe,
        ↪  bel_mksymbol("chars"));
11
12      int i;
13
14      // Get nth cdr
15      for(i = 0; i < first_char; i++) {
16          bel = bel_cdr(bel);
17      }
18
19      i = 'a';
20      while(!bel_nilp(bel) && i < first_char + 10) {
21          Bel *car = bel_car(bel);
22          printf("Char: %03d (%c) => ",
23                 bel_car(car)->chr,
24                 ((Bel_char)i));
25          bel_print(bel_cdr(car));
26          putchar(10);
27          bel = bel_cdr(bel);
28          i++;
29      }
30  }
```

Read file bit by bit

This test opens up the Believe C source code file as a read stream, using Bel's stream structure, then proceeds to read ten bytes from it (meaning that it will read 80 bits). Every eight bit will be stored in a Bel list and then converted to a proper Bel character, which will be displayed on screen along with its bits.

It is interesting to notice that, since the bit-reading operation itself returns characters `\0` or `\1`, the bit list composing a character is always a Bel string.

```
1   void
2   read_file_test()
3   {
4       // We are going to read ten bytes from Bel's
5       // own source code file.
6       Bel *file = bel_mkstream("believe.c", BEL_STREAM_READ);
7
8       if(bel_errorp(file)) {
9           bel_print(file);
10          return;
```

```
11      }
12
13      printf("Stream: ");
14      bel_print(file);
15      putchar(10);
16
17      int n_bytes = 10;
18      while(n_bytes > 0) {
19          // 1 byte = 8 bits, so we make a list of
20          // eight characters
21          Bel **char_nodes = GC_MALLOC(8 * sizeof(Bel*));
22
23          int i;
24          for(i = 0; i < 8; i++) {
25              Bel *read_char =
26                  bel_stream_read_bit(&file->stream);
27              char_nodes[i] = bel_mkpair(read_char, bel_g_nil);
28          }
29
30          // Link nodes
31          for(i = 0; i < 7; i++) {
32              char_nodes[i]->pair->cdr = char_nodes[i + 1];
33          }
34
35          // Display on screen
36          bel_print(char_nodes[0]);
37          printf(" => ");
38          bel_print(
39              bel_char_from_binary(char_nodes[0]));
40          putchar(10);
41
42          n_bytes--;
43      }
44
45      bel_stream_close(file);
46  }
```

Display errors

We generate a few errors and grab them, then we print these errors on screen to show their literal structure.

```
1  void
2  show_errors_test()
```

```
3  {
4      Bel *err;
5
6      // Unexisting file
7      err = bel_mkstream("waddawaddawadda", BEL_STREAM_READ);
8      bel_print(err);
9      putchar(10);
10     printf("Is this an error? %c\n",
11            bel_errorp(err) ? 'y' : 'n');
12
13     // Incorrect use of car and cdr
14     err = bel_car(bel_g_t);
15     bel_print(err); putchar(10);
16     err = bel_cdr(bel_g_t);
17     bel_print(err); putchar(10);
18
19     // Incorrect generation of Bel character from binary
20     /* Bel *str = bel_mkstring("110"); */
21     /* err = bel_char_from_binary(str); */
22     /* bel_print(err); putchar(10); */
23
24     /* str = bel_mkstring("110a1101"); */
25     /* err = bel_char_from_binary(str); */
26     /* bel_print(err); putchar(10); */
27 }
```

Lookup primitives

We look up a few registered primitives in the global environment, and print them in their literal form.

```
1  void
2  lookup_primitives_test()
3  {
4      Bel *bel;
5      bel = bel_lookup(bel_g_nil, bel_mksymbol("car"));
6      bel_print(bel);
7      putchar(10);
8
9      bel = bel_lookup(bel_g_nil, bel_mksymbol("cdr"));
10     bel_print(bel);
11     putchar(10);
12
13     bel = bel_lookup(bel_g_nil, bel_mksymbol("coin"));
```

```
14      bel_print(bel);
15      putchar(10);
16
17      bel = bel_lookup(bel_g_nil, bel_mksymbol("stat"));
18      bel_print(bel);
19      putchar(10);
20
21      // Undefined primitive
22      bel_print(bel_g_nil); putchar(10);
23      bel = bel_lookup(bel_g_nil, bel_mksymbol("wadawada"));
24      bel_print(bel);
25      putchar(10);
26  }
```

Environment tests

The first test involves creating a temporary lexical environment, pushing a few literals, assigning values, unbinding values too.

```
1   void
2   lexical_environment_test()
3   {
4       Bel *lexenv = bel_g_nil;
5       Bel *ret;
6
7       puts("    -- Registering local `foo`");
8       lexenv = bel_env_push(lexenv,
9                             bel_mksymbol("foo"),
10                            bel_mksymbol("bar"));
11
12      printf("Environment:       ");
13      bel_print(lexenv);
14      printf("\nLookup:            ");
15      bel_print(bel_lookup(lexenv, bel_mksymbol("foo")));
16      putchar(10); putchar(10);
17
18      // Assignment
19      puts("    -- Assigning new value to `foo`");
20      ret =
21          bel_assign(
22              lexenv,
23              bel_mksymbol("foo"),
24              bel_mkliteral(
25                  bel_mkpair(bel_mksymbol("baz"),
```

```
26                              bel_g_nil)));
27
28      printf("Environment:       ");
29      bel_print(lexenv);
30      printf("\nAssignment result: ");
31      bel_print(ret);
32      printf("\nLookup:            ");
33      bel_print(bel_lookup(lexenv, bel_mksymbol("foo")));
34      putchar(10); putchar(10);
35
36      // Unbinding
37      puts("    -- Unbinding `foo`");
38      ret = bel_unbind(&lexenv, bel_mksymbol("foo"));
39
40      printf("Environment:       ");
41      bel_print(lexenv);
42      printf("\nUnbinding result:  ");
43      bel_print(ret);
44      printf("\nLookup:            ");
45      bel_print(bel_lookup(lexenv, bel_mksymbol("foo")));
46      putchar(10);
47  }
```

Second test is creating a global variable through assignment, creating a variable bound to the same symbol on a lexical environment, unbinding both, then performing a last invalid unbinding.

```
1   void
2   global_assignment_test()
3   {
4       Bel *lexenv = bel_g_nil;
5       Bel *ret;
6
7       // Global creation through assignment
8       puts("    -- Assigning `foo` without previous definition");
9       ret = bel_assign(bel_g_nil,
10                       bel_mksymbol("foo"),
11                       bel_mksymbol("bar"));
12
13      printf("Assignment result: ");
14      bel_print(ret);
15      printf("\nLookup:            ");
16      bel_print(bel_lookup(bel_g_nil, bel_mksymbol("foo")));
17      putchar(10); putchar(10);
```

```
18
19      // Local creation of variable bound to
20      // same symbol
21      puts("    -- Shadowing global `foo` with a local");
22      lexenv =
23          bel_env_push(lexenv,
24                       bel_mksymbol("foo"),
25                       bel_mksymbol("quux"));
26
27      printf("Environment:       ");
28      bel_print(lexenv);
29      printf("\nLookup:            ");
30      bel_print(bel_lookup(lexenv, bel_mksymbol("foo")));
31
32      // Three unbindings
33      printf("\n    -- Unbinding `foo` three times");
34      int i;
35      for(i = 0; i < 3; i++) {
36          ret = bel_unbind(&lexenv, bel_mksymbol("foo"));
37
38          printf("\n\n      After unbinding.");
39          printf("\nEnvironment:       ");
40          bel_print(lexenv);
41          printf("\nUnbinding result:  ");
42          bel_print(ret);
43          printf("\nLookup:            ");
44          bel_print(bel_lookup(lexenv, bel_mksymbol("foo")));
45      }
46      putchar(10);
47  }
```

Number test

This test performs raw arithmetic on the four subtypes of numbers: integers, floats, complexes and fractions.

```
1   void
2   number_test()
3   {
4       Bel *a;
5       Bel *b;
6
7       // Integer sum
8       a = bel_mkinteger(4);
```

```
9       b = bel_mkinteger(2);
10
11      bel_print(a);
12      printf(" + ");
13      bel_print(b);
14      printf(" = ");
15      bel_print(bel_num_add(a, b));
16      putchar(10);
17
18
19      // Float subtraction
20      a = bel_mkfloat(4.0);
21      b = bel_mkfloat(3.5);
22
23      bel_print(a);
24      printf(" - ");
25      bel_print(b);
26      printf(" = ");
27      bel_print(bel_num_sub(a, b));
28      putchar(10);
29
30
31      // Fraction sum
32      a = bel_mkfraction(bel_mkinteger(1),
33                         bel_mkinteger(3));
34      b = bel_mkfraction(bel_mkinteger(1),
35                         bel_mkinteger(6));
36
37      bel_print(a);
38      printf(" + ");
39      bel_print(b);
40      printf(" = ");
41      bel_print(bel_num_add(a, b));
42      putchar(10);
43
44
45      // Complex multiplication
46      a = bel_mkcomplex(bel_mkinteger(3),
47                        bel_mkinteger(2));
48      b = bel_mkcomplex(bel_mkinteger(1),
49                        bel_mkinteger(4));
50
51      bel_print(a);
```

```
52      printf(" * ");
53      bel_print(b);
54      printf(" = ");
55      bel_print(bel_num_mul(a, b));
56      putchar(10);
57
58      // Complex division
59      // Reusing a and b from last example
60      bel_print(a);
61      printf(" / ");
62      bel_print(b);
63      printf(" = ");
64      bel_print(bel_num_div(a, b));
65      putchar(10);
66
67      // Integer division (inexact)
68      a = bel_mkinteger(7);
69      b = bel_mkinteger(2);
70
71      bel_print(a);
72      printf(" / ");
73      bel_print(b);
74      printf(" = ");
75      bel_print(bel_num_div(a, b));
76      putchar(10);
77  }
```

Debriefing macro

This macro is a helper for debriefing results of evaluation tests.

```
1  #define BEL_EVAL_DEBRIEF(exp, res, env)  \
2      {                                    \
3      printf("Expression: ");              \
4      bel_print(exp); putchar(10);         \
5      res = bel_eval(exp, env);            \
6      printf("Result: ");                  \
7      bel_print(res); putchar(10);         \
8      putchar(10);                         \
9      }
```

Evaluator test

This test performs the evaluation of a few forms so that we can check if the evaluator runs properly.

```
1  void
2  eval_test()
3  {
4      Bel *form;
5      Bel *result;
6
7      // (quote foo)
8      form = bel_mklist(2,
9                        bel_mksymbol("quote"),
10                       bel_mksymbol("foo"));
11     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
12
13     // (join (quote foo) (quote bar))
14     form =
15         bel_mklist(
16             3,
17             bel_mksymbol("join"),
18             bel_mklist(
19                 2,
20                 bel_mksymbol("quote"),
21                 bel_mksymbol("foo")),
22             bel_mklist(
23                 2,
24                 bel_mksymbol("quote"),
25                 bel_mksymbol("bar")));
26     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
27
28
29     // (fn (x) (id x x))
30     form = bel_mklist(
31         3,
32         bel_mksymbol("fn"),
33         bel_mklist(1, bel_mksymbol("x")),
34         bel_mklist(
35             3,
36             bel_mksymbol("id"),
37             bel_mksymbol("x"),
38             bel_mksymbol("x")));
39     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
```

```
40
41
42      // ((fn (x) (id x x)) (quote foo))
43      form = bel_mklist(
44          2,
45          form, // Use closure from last example
46          bel_mklist(2,
47                     bel_mksymbol("quote"),
48                     bel_mksymbol("foo")));
49      BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
50
51
52      // (if (id (quote bar) (quote foo)) (quote okay)
53      //     (id (quote foo) (quote bar)) (quote okay)
54      //                                  (quote nope))
55      form =
56          bel_mklist(
57              6,
58              bel_mksymbol("if"),
59              // Clause 1
60              bel_mklist(
61                  3,
62                  bel_mksymbol("id"),
63                  bel_mklist(
64                      2,
65                      bel_mksymbol("quote"),
66                      bel_mksymbol("bar")),
67                  bel_mklist(
68                      2,
69                      bel_mksymbol("quote"),
70                      bel_mksymbol("foo"))),
71              bel_mklist(
72                  2,
73                  bel_mksymbol("quote"),
74                  bel_mksymbol("okay")),
75              // Clause 2
76              bel_mklist(
77                  3,
78                  bel_mksymbol("id"),
79                  bel_mklist(
80                      2,
81                      bel_mksymbol("quote"),
82                      bel_mksymbol("foo")),
```

```
83                  bel_mklist(
84                      2,
85                      bel_mksymbol("quote"),
86                      bel_mksymbol("bar"))),
87              bel_mklist(
88                  2,
89                  bel_mksymbol("quote"),
90                  bel_mksymbol("okay")),
91              // Alternative
92              bel_mklist(
93                  2,
94                  bel_mksymbol("quote"),
95                  bel_mksymbol("nope")));
96      BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
97
98
99      // (sys "echo Hello, world!")
100     // NOTE: I am commenting out this test since
101     //       this function could open some security
102     //       holes in systems unadvertedly using it.
103     /* form = bel_mklist( */
104     /*     2, */
105     /*     bel_mksymbol("sys"), */
106     /*     bel_mkstring("echo Hello, world!")); */
107     /* BEL_EVAL_DEBRIEF(form, result, bel_g_nil); */
108
109
110     // Eval some axioms
111     puts("Evaluating some axioms");
112     form = bel_g_t;
113     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
114
115     form = bel_g_o;
116     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
117
118     form = bel_g_apply;
119     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
120
121     form = bel_g_nil;
122     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
123
124
125     // Eval some numbers
```

```
126     form = bel_mkinteger(42);
127     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
128
129     form = bel_mkfloat(42.0);
130     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
131
132     form = bel_mkfraction(bel_mkinteger(2),
133                           bel_mkinteger(3));
134     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
135
136
137     form = bel_mkcomplex(bel_mkfloat(2.0),
138                          bel_mkfloat(3.4));
139     BEL_EVAL_DEBRIEF(form, result, bel_g_nil);
140 }
```

Arithmetic evaluation test

This next test tests the evaluation of arithmetic on some numbers, from calls to the evaluator itself.

```
1  void
2  arithmetic_eval_test()
3  {
4      Bel *exp;
5      Bel *result;
6
7      // (+ 2 #(c 3+7i) #(f 1/3))
8      exp = bel_mklist(
9          4,
10         bel_mksymbol("+"),
11         bel_mkinteger(2),
12         bel_mkcomplex(bel_mkinteger(3),
13                       bel_mkinteger(7)),
14         bel_mkfraction(bel_mkinteger(1),
15                        bel_mkinteger(3)));
16     BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
17
18     // (id #(c 1+3i) #(c 1+3i))
19     exp = bel_mklist(
20         3,
21         bel_mksymbol("id"),
22         bel_mkcomplex(bel_mkinteger(1),
23                       bel_mkinteger(3)),
```

```
24          bel_mkcomplex(bel_mkinteger(1),
25                        bel_mkinteger(3)));
26      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
27
28
29      //(- #(c 3-8i))
30      exp = bel_mklist(
31          2,
32          bel_mksymbol("-"),
33          bel_mkcomplex(bel_mkinteger(3),
34                        bel_mkinteger(8)));
35      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
36
37
38      // (* 1 2 3 4 5)
39      exp = bel_mklist(
40          6,
41          bel_mksymbol("*"),
42          bel_mkinteger(1),
43          bel_mkinteger(2),
44          bel_mkinteger(3),
45          bel_mkinteger(4),
46          bel_mkinteger(5));
47      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
48
49      // (/ 45.0)
50      exp = bel_mklist(
51          2,
52          bel_mksymbol("/"),
53          bel_mkfloat(45.0));
54      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
55
56
57      // Spec conformity tests
58      // (-) should return 0
59      exp = bel_mklist(1, bel_mksymbol("-"));
60      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
61
62      // (/) should return 1
63      exp = bel_mklist(1, bel_mksymbol("/"));
64      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
65
66      // (/ 5) should return 5
```

```
67      exp = bel_mklist(
68          2,
69          bel_mksymbol("/"),
70          bel_mkinteger(5));
71      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
72  }
```

Arity tests

The following tests check for the arity of primitive functions. By default, a small number of arguments is not a bug, and the missing arguments are traded for `nil`.

```
1   void
2   arity_test()
3   {
4       Bel *exp;
5       Bel *result;
6
7       // (id) => t
8       exp = bel_mklist(1, bel_mksymbol("id"));
9       BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
10
11      // (join) => (nil . nil)
12      exp = bel_mklist(1, bel_mksymbol("join"));
13      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
14
15      // (type) => symbol
16      exp = bel_mklist(1, bel_mksymbol("type"));
17      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
18  }
```

Dynamic binding test

This test attempts to bind the result of the division between `1` and `2` to a dynamic variable `x`, then proceeds to perform an operation with it.

```
1   void
2   dynamic_binding_test()
3   {
4       Bel *exp;
5       Bel *result;
6
7       // (dyn x (/ 1 2)
8       //   (+ x 1))
```

```
9       exp = bel_mklist(
10          4,
11          bel_mksymbol("dyn"),
12          bel_mksymbol("x"),
13          bel_mklist(
14              3,
15              bel_mksymbol("/"),
16              bel_mkinteger(1),
17              bel_mkinteger(2)),
18          bel_mklist(
19              3,
20              bel_mksymbol("+"),
21              bel_mksymbol("x"),
22              bel_mkinteger(1)));
23      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
24  }
```

Global binding test

This test attributes a certain closure to the symbol `square` globally, then proceeds to apply this new global function to some number.

```
1   void
2   global_binding_test()
3   {
4       Bel *exp;
5       Bel *result;
6
7       // function definition
8       // (fn (x) (* x x))
9       exp =
10          bel_mklist(
11              3,
12              bel_mksymbol("fn"),
13              bel_mklist(1, bel_mksymbol("x")),
14              bel_mklist(
15                  3,
16                  bel_mksymbol("*"),
17                  bel_mksymbol("x"),
18                  bel_mksymbol("x")));
19
20      // assignment
21      // (set square (fn (x) (* x x)))
22      exp = bel_mklist(
```

```
23          3,
24          bel_mksymbol("set"),
25          bel_mksymbol("square"),
26          exp); // Reuse defined function
27      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
28
29      // Type check
30      // (type square)
31      exp = bel_mklist(
32          2,
33          bel_mksymbol("type"),
34          bel_mksymbol("square"));
35      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
36
37      // (square #(f 1/2))
38      exp = bel_mklist(
39          2,
40          bel_mksymbol("square"),
41          bel_mkfraction(bel_mkinteger(1),
42                         bel_mkinteger(2)));
43      BEL_EVAL_DEBRIEF(exp, result, bel_g_nil);
44
45      // TODO: Unintern symbol?
46  }
```

Basic tokenizer test

This tests the tokenize. Basically, it will take any string expression and attempt to generate a list of Bel strings, each string being a token.

We begin by declaring a helper macro to reduce code repetition.

```
1  #define BEL_TOKENIZE_DEBRIEF(exp, res, str)              \
2      {                                                    \
3          exp = str;                                       \
4          res = bel_tokenize(exp);                         \
5          printf("Expression:\n%s\nResult: ", exp);        \
6          bel_print(res);                                  \
7          putchar(10);                                     \
8      }
```

Now we write the test. This takes the exp variable, the result variable, and a potentially multiline string. Each case is a tokenization test.

```
1  void
2  basic_tokenizer_test()
```

```
3  {
4      const char *exp;
5      Bel *result = bel_g_nil;
6
7      BEL_TOKENIZE_DEBRIEF(exp, result, "(+ 1 2)");
8
9      BEL_TOKENIZE_DEBRIEF(
10         exp, result,
11         "(progn (+ 1 2)\n"
12         "        (+ 3 4))");
13
14     BEL_TOKENIZE_DEBRIEF(
15         exp, result,
16         "(def some (f xs)\n"
17         "  (if (no xs)      t\n"
18         "      (f (car xs)) (all f (cdr xs))\n"
19         "                   nil))");
20
21 }
```

Basic parser test

This is a test for the parser. This takes the tokenizer's output and attempts to parse each token into a proper Bel object described by it.

We begin with a macro much like the one for the tokenizer, but here, we perform tokenization and then parsing.

```
1  #define BEL_PARSER_DEBRIEF(exp, res, str)                \
2      {                                                    \
3          exp = bel_tokenize(str);                         \
4          res = bel_parse_expr(exp, 0);                    \
5          printf("Expression:\n" str "\nResult: ");        \
6          bel_print(res);                                  \
7          putchar(10);                                     \
8      }
```

The test itself just invokes the macro, and takes the exact same variables needed by the tokenizer tests. Here, instead of returning a flat list of strings, we return a list of all parsed expressions, be them lists or not.

```
1  void
2  parser_test()
3  {
4      Bel *expr;
```

```
5       Bel *result;
6
7       BEL_PARSER_DEBRIEF(expr, result, "(+ 1 2)");
8
9       BEL_PARSER_DEBRIEF(
10          expr, result,
11          "(progn (+ 1 2)\n"
12          "       (+ 2 3))");
13
14      BEL_PARSER_DEBRIEF(
15          expr, result,
16          "(def some (f xs)\n"
17          "  (if (no xs)      t\n"
18          "      (f (car xs)) (all f (cdr xs))\n"
19          "                   nil))");
20  }
```

Arbitrary input parsing

This test takes an arbitrary input from the user and parses it into proper Bel objects.

As the behavior suggests, this is extremely unsafe! But it can be used in debug occasions. Plus, this uses `fgets` for user input, which has its constraints on buffer overflow, but does not offer the same guarantees as `libeditline` or `libreadline`.

When you are done, type `#q` to finish the parsing session.

```
1   void
2   arbitrary_input_parsing()
3   {
4       Bel *result;
5       Bel *tokens;
6       char input[1024] = {0};
7       puts("When you are done, type #q to exit.");
8       puts("And don't worry about flush errors.");
9       while(1) {
10          printf("parse> ");
11          fgets(input, 1024, stdin);
12          input[strlen(input)] = '\0';
13
14          // Sorry about that :V
15          if(input[0] == '#' &&
16             input[1] == 'q')
17              break;
18
19          tokens = bel_tokenize(input);
```

```
20          puts("Tokens:");
21          bel_print(tokens); putchar(10);
22
23          result = bel_parse_expr(tokens, 0);
24          puts("Result:");
25          bel_print(result); putchar(10);
26      }
27  }
```

Test-only REPL

This implements a test REPL which takes an input from the user and attempts to evaluate everything that is written, and also works as an entry point where the tests can be accessed as well, by typing the command `#t`.

This is supposed to test the parser and the evaluator together, those not being the interpreter itself.

This is also very unsafe. There are a lot of checks we are not performing here, and those are on purpose, since this is for debug and testing purposes only.

When you are done, type `#q` to finish the evaluation session.

```
1   void run_tests();
2
3   void
4   test_repl()
5   {
6       Bel *result;
7       Bel *tokens;
8       char input[1024] = {0};
9       puts("Enter an expression to be evaluated, or type #t for
        ↪  the test menu.");
10      puts("When you are done, type #q to exit.");
11      while(1) {
12          printf("> ");
13          fgets(input, 1024, stdin);
14          input[strlen(input)] = '\0';
15
16          // Sorry about that
17          if(input[0] == '#') {
18              if (input[1] == 'q')
19                  break;
20              else if (input[1] == 't') {
21                  run_tests();
22                  break;
23              }
```

```
24          }
25
26          tokens = bel_tokenize(input);
27          result = bel_parse_expr(tokens, 0);
28          if(!bel_errorp(result)) {
29              result = bel_eval(bel_car(result),
30                                bel_g_nil);
31          }
32          bel_print(result); putchar(10);
33      }
34  }
```

CHAPTER 12

Entry point

12.1 Initialization

This is the initialization function for the Bel interpreter. Once this function is called, the Bel system is ready to be used.

```
1  Bel*
2  bel_init(void)
3  {
4      // Initialize garbage collector
5      GC_INIT();
6
7      // Initialize random number generation
8      // Warning: This is a VERY naive approach
9      srand(time(NULL));
10
11     // Initialize symbol table
12     bel_sym_table_init();
13
14     // Axioms
15     bel_init_ax_vars();
16     bel_init_ax_chars();
17     bel_init_streams();
18     bel_init_ax_env();
19     bel_init_ax_primitives();
20
21     // Read macros
22     bel_init_read_macros();
23
```

```
24      // TODO: Return an environment?
25      return bel_g_nil;
26  }
```

12.2 Tests

This is the entry point for tests. All running tests are to be put here.

We also make sure that these tests are run as a menu, so that only the desired test is shown when needed.

```
1   void
2   run_tests()
3   {
4       int opt;
5
6       do {
7           puts("-- Believe test menu\n"
8                "   Choose a test to run:\n"
9                " 1. String test\n"
10               " 2. Notation test\n"
11               " 3. List test\n"
12               " 4. Closure representation test\n"
13               " 5. Character List & Lookup test\n"
14               " 6. Read five bytes from Believe's source\n"
15               " 7. Show a few errors on screen\n"
16               " 8. Lookup a few primitives and print them\n"
17               " 9. Lexical environment test\n"
18               "10. Globals and assignment tests\n"
19               "11. Number arithmetic tests\n"
20               "12. Evaluator test\n"
21               "13. Arithmetic evaluation test\n"
22               "14. Primitive arity test\n"
23               "15. Dynamic binding test\n"
24               "16. Global binding test\n"
25               "17. Basic tokenizer test\n"
26               "18. Parser test\n"
27               "19. Arbitrary input parser (unsafe!)\n"
28               "20. Go back to REPL\n"
29
30               " 0. Exit menu");
31          printf("Option >> ");
32          scanf("%d", &opt);
33
```

```
34          // flush here
35
36          putchar(10);
37          switch(opt) {
38          default: puts("Invalid option.");    break;
39          case  0:  break;
40          case  1: string_test();              break;
41          case  2: notation_test();            break;
42          case  3: list_test();                break;
43          case  4: closure_repr_test();        break;
44          case  5: character_list_test();      break;
45          case  6: read_file_test();           break;
46          case  7: show_errors_test();         break;
47          case  8: lookup_primitives_test();   break;
48          case  9: lexical_environment_test(); break;
49          case 10: global_assignment_test();   break;
50          case 11: number_test();              break;
51          case 12: eval_test();                break;
52          case 13: arithmetic_eval_test();     break;
53          case 14: arity_test();               break;
54          case 15: dynamic_binding_test();     break;
55          case 16: global_binding_test();      break;
56          case 17: basic_tokenizer_test();     break;
57          case 18: parser_test();              break;
58          case 19: arbitrary_input_parsing();  break;
59          case 20:
60              test_repl();
61              opt = 0;
62              break;
63          }
64      } while(opt != 0);
65  }
```

12.3 `main` function

This is the program entry point. It is supposed to only print the ribbon, initialize Bel and perform some tests, for now.

```
1   int
2   main(void)
3   {
4       printf("Believe %s (built %s)\n",
5              BELIEVE_VERSION,
```

```
6             BELIEVE_BUILD_TIME);
7     printf("A Bel Lisp interpreter\n");
8     printf("Copyright (c) %s\n", BELIEVE_COPYRIGHT);
9     printf("This software is distributed under the %s
      ↪  license.\n",
10          BELIEVE_LICENSE);
11
12    bel_init();
13    test_repl();
14
15    return 0;
16 }
```