A literate implementation in C of the Categorical Abstract Machine

Arno Bastenhof

February 5, 2017

Contents

Chapter 1

Introduction

The current document details a modest literate implementation in the C programming language of an abstract machine for interpreting functional programming languages. The practice of literate programming has originated in a proposal by Knuth [10], and we will have a few more words to say about it below. Abstract machines for functional programs, in turn, go back to Landin's SECD [12] (named after its components, the *Stack*, *Environment*, *Control* and *Dump*), originally presented as a means of implementing ISWIM [13]. Here, however, we shall instead focus on the more recent *Categorical Abstract Machine* (or *CAM*, for short) from Cousineau, Curien and Mauny [1], having its roots in De Bruijn's name-free notation for binding constructs [5], Curry et al.'s combinatory logic [4] and category theory (an introductory account of which from the perspective of computer science may be found in, e.g., [2]).

We shall proceed as follows. First, in §1.1 a brief introduction will be provided on the (untyped) λ-calculus, serving as the core of any functional programming language. Next, §1.2 discusses literate programming and our implementation of it, with the works of Fraser and Hanson [6], [7] constituting our main sources of inspiration. Our coding style and conventions are documented in §1.3, again taking more than a hint or two from Fraser and Hanson. Finally, in §1.4 the scope of this work is dicussed, together with an outline of the project as a whole. Our aims having been primarily self-educational, we imposed severe restrictions from the outset on the results we intended to achieve in order to maintain focus and allow for completion within a reasonable timeframe. Most notably, our interest was primarily in the abstract machine itself, and the frontend we wrote for it was kept as simple as possible to the extent of having become useless beyond providing access to the CAM's internals. The intent of this project was not to learn about Hindley-Milner type inference or language design, and it shows.

1.1 λ-calculus

The primary syntactic unit around which much of mathematical discourse revolves is arguably that of an expression. We encounter them in such simple algebraic compositions as

$$(y - 2) + z$$

ranging to the more involved musings of calculus and beyond,

$$\sum_{n=0}^{\infty} r^n$$

$$\int_a^b x^2 dx$$

Already from this small number of examples, we can make several observations:

- Expressions are built by applying operators like $+$ or $\sum$ to operands, the latter expressions themselves. Fueling this recursion are any constants, like numbers, or, leading into our next two observations, variables.
- Expressions may contain free variables, like a, b, z, y and r above, usually interpreted within the wider context as being universally quantified.
- Expressions may contain bound variables, like n and x above. In fact, free variables may be considered bound as well by the implicitly understood universal quantifiers at the level of the surrounding discourse.

Variable binding and the operator/operand dichotomy pervade mathematical notation, and a closer analysis of their subtleties will yield conclusions that similarly apply across the board. For instance, what are we to make of two expressions differing only by a renaming of some bound variable? E.g., as with $\sum_{i=1}^{k} i$ and $\sum_{j=1}^{k} j$? The distinction certainly seems inconsequential, but then how about $\sum_{i=1}^{2} \sum_{j=1}^{3} i + j$ and $\sum_{j=1}^{2} \sum_{j=1}^{3} j + j$? The second expression arises from the former by renaming the variable bound by the inner sum, but the results of their evaluation differ. The point is that we shall want to address these and similar questions once and for all, as opposed to returning to them time and again whenever some new variable-binding construct is proposed. Put more ambituously, we shall seek a method of analysis that reduces any expression to the application of a small number of primitive notions, on the basis of which such questions as raised above may be fruitfully investigated independently of any one particular branch of applied mathematics.

The challenge put forth in the previous paragraph was met by Alonzo Church in the 1930's with his invention of the λ-calculus (a recent survey of which may be found in Sørensen et al. [15]), named after the notational device it uses for denoting variable bindings. Roughly, Church put forth a notion of expression that we shall henceforth refer to by *terms*, and which in themselves suffice for subsuming much of the more commonly encountered writings in math.

Proceeding informally, terms, together with the closely related concepts of free and bound variable occurrences, may be defined by induction as follows:

0. Any *constant* is a term, where by a constant we may understand not only, e.g., symbolic representations of numbers, but also such writings as commonly associated with operators, like, e.g., $+$, $\sum$, $\int$, etc.

1. Assuming some countably infinite voculary of symbols distinct from those used for constants, every *variable* x drawn therefrom is a term, with any such occurrence by itself being considered free.

2. If t, s are terms, so is their *application* $(t\ s)$, with all free variable occurrences in s, t separately likewise occurring free in $(s\ t)$.

3. If t is a term and x a variable, so the *abstraction* $(\lambda x.t)$ is a term, where all free occurrences of x in t are now deemed bound in $(\lambda x.t)$.

Before demonstrating the practical application of these definitions, we first draw attention to our emphasis on free- and bound *occurrences* of variables. For instance, in

$$((\lambda x.x)\ x)$$

the variable x has two occurrences outside of its appearance right after λ, with one being bound and the other free.

The key to understanding how our previous expressions from algebra and calculus may be represented by terms lies in the realization that in λ-calculus, traditional operators are treated as constants, rendering the combination with operands through (repeated) application, while reducing all forms of variable binding to abstraction. To illustrate,

$$\begin{aligned} (y-2)+z &\sim ((+\ (-\ y\ 2))\ z) \\ \textstyle\sum_{n=0}^{\infty} r^n &\sim (((\sum\ 0)\ \infty)\ (\lambda n.((\uparrow\ r)\ n))) \\ \textstyle\int_a^b f(x)dx &\sim (((\int\ a)\ b)\ (\lambda x.f(x))) \end{aligned}$$

If we agree that applications associate to the left, and ommitting the brackets surrounding abstractions wherever this does not lead to confusion, these may be rendered somewhat more intelligibly:

$$(+\ (-\ y\ 2)\ z)$$

$$(\sum\ 0\ \infty\ \lambda n.(\uparrow\ r\ n))$$

$$(\int\ a\ b\ \lambda x.f(x))$$

Landin [12] offers many more illustrations of how familiar constructs from both the languages of mathematics and of computer science may be adequately rendered by λ-terms, discussing as well criteria for judging the faithfulness of such translations. In the sections to follow we shall encounter several more of Landin's

examples, though for now the above introduction should suffice. Our main concern in the sequel, however, shall be the *evaluation* of terms, i.e., a process similar to that by which, say, the expression $1 + 1$ from arithmetic yields 2. In fact, the very purpose of the abstract machine proposed by Cousineau et al. is precisely to formulate such a procedure that applies to λ-terms in general, and hence to all specific expressions that may be encoded thereby, be they from logic, programming, calculus, etc.

1.2 Literate programs

The current document constitutes a *literate program*, in the sense that it has been generated from the same sources containing the code that it describes. Literate programming was introduced by Knuth [10], who called for a shift in thinking about the writing of code towards the art of explaining others what we desire the computer to do, as opposed to a mere isolated interaction with the machine. For our own purposes, we have embraced this practice as a method of study. It has often been said that true understanding is contingent on the ability to explain. As such, in desiring to learn some new topic, we might do well to articulate the evolution in our thought processes along the way, thus forcing ourselves to mentally deconstruct whichever narrative we have chosen to serve as our study materials, and rebuilding it from the ground up in a way that seems most meaningful to ourselves. The act of focus required by such a process will no doubt go a long way towards attainment of the understanding that we seek.

Various toolings have been developed for enabling literate programming, there for a long time having existed a tight coupling with individual languages. This changed with the introduction of NOWEB by Ramsey [14], fueled by the design goal of offering a simple language-agnostic interface that would not get in the way of the programmer. NOWEB has since been applied in various projects, among which a production-quality C compiler [6] and a general-purpose library for the C programming language [7]. The latter two sources having served as our main source of inspiration in composing this document, the choice for NOWEB was easily made.

For the remainder of this section, we shall discuss some of the capabilities offered by literate programming. Its primary device is the ability to present code in a different order than required for consumption by a compiler, born from an observed discrepancy with the optimal sequencing of thoughs typically required by the human mind for attaining comprehension. The author is thus free to reshuffle the contents of his program in whichever way best aids his explanations. To illustrate, consider the task of writing a lexer for some programming language. Typically, these consist of a single big hairy method for requesting the next token from an input stream. To keep things simple, let's say we are parsing expressions from arithmetic, built from integers using the operations of addition, multiplication, subtraction and division. We shall first create a header for collecting all public declarations. As is common practice, we will use internal

include guards to prevent it from being included more than once.

8 ⟨*lex.h* 8⟩≡
```
#ifndef LEX_H_
#define LEX_H_

⟨lex.h typedefs 9a⟩
⟨lex.h function prototypes 9b⟩

#endif /* LEX_H_ */
```

A literate program, in NOWEB's view, is a collection of *chunks*, be they *code* chunks, like the above, or *documentation* chunks, like the current paragraph. Each code chunk has a name, such as *lex.h*, and may mention other code chunks, e.g., *lex.h typedefs*, or *lex.h function prototypes*.

We need distinguish but a few token types, to wit (say, integral) numbers, brackets, and the usual arithmetic operations. In addition, a sentinel is added to the mix for signaling the end of the input stream. We capture these ideas in a type definition, filling in the first of the referenced code chunks in *lex.h*.

9a ⟨*lex.h typedefs* 9a⟩≡ (8)
```
typedef enum {
  EOS = 0,      /* end of stream */
  INT = 1,      /* integer */
  LBRACK = 40,  /* '(' */
  RBRACK = 41,  /* ')' */
  MULT = 42,    /* '*' */
  PLUS = 43,    /* '+' */
  MINUS = 45,   /* '-' */
  DIV = 47      /* '/' */
} type_t;
```

Note that except for `INT` and `EOS`, all types describe single-character strings. To make our lives easier later, we have defined these by the corresponding ASCII values. Moving on to the interface for getting the next token, we shall again simplify matters by ignoring the matched token string and returning only the recognized type, as well as by assuming the input stream to coincide with standard input.

9b ⟨*lex.h function prototypes* 9b⟩≡ (8)
```
extern type_t GetToken(void);
```

With our interface clear, we can proceed to its implementation.

9c ⟨*lex.c* 9c⟩≡
```
⟨lex.c includes 10a⟩
⟨lex.c function definitions 10c⟩
```

In practice, we shall always reserve the topmost include in a `.c` file for the associated header, thus ensuring the latter is self-contained.

10a ⟨*lex.c includes* 10a⟩≡ (9c) 10b▷
```
#include "lex.h"
```

This brings us to the main attraction. `GetToken` tries to determine the type of token that it is reading based on the first non-whitespace character, while signalling an error if unsuccessful.

10b ⟨*lex.c includes* 10a⟩+≡ (9c) ◁10a 11b▷
```
#include <ctype.h>
```

10c ⟨*lex.c function definitions* 10c⟩≡ (9c)
```
type_t
GetToken(void)
{
  int c;  /* next input character */

  /* skip whitespace */
  while (isspace(c = getchar()))
    ;

  switch (c) {
  ⟨lex.c cases 10d⟩
  default:
    ⟨lex.c error handling 11c⟩
  }
}
```

Note that the first of the code chunks above shares its name with another one defined before. As indicated by the additional + symbol, we interpret the second one as being appended to the first.

Before getting into the more serious cases, let us not forget to handle the end of the input stream.

10d ⟨*lex.c cases* 10d⟩≡ (10c) 10e▷
```
case EOF:
  return EOS;
```

Recall that we defined those token types describing single characters by the latter's ASCII value, motivating the following straightforward implementation:

10e ⟨*lex.c cases* 10d⟩+≡ (10c) ◁10d 11a▷
```
case '+': case '-': case '*': case '/': case '(': case ')':
  return c;
```

This still leaves the recognition of integers:

11a ⟨*lex.c cases* 10d⟩+≡ (10c) ◁10e
```
case '0': case '1': case '2': case '3': case '4': case '5':
case '6': case '7': case '8': case '9':
  while (isnum(c = getchar()))
    ;
  if (c != EOF) {
    ungetc(c, stdin);
  }
  return INT;
```

Note that the last character read will be one too many if not `EOF`, requiring us to push it back on the input stream. Finally, if the first non-whitespace character that we read is not already covered by the above cases, we print an error message and crash hard, not wishing to bother with proper error handling for this simple example:

11b ⟨*lex.c includes* 10a⟩+≡ (9c) ◁10b
```
#include <stdio.h>
```

11c ⟨*lex.c error handling* 11c⟩≡ (10c)
```
fprintf(stderr, "Unexpected character: %c\n", c);
exit(1);
```

1.3 Coding conventions

The web abounds with coding conventions for the C programming language, there seemingly being no de-facto standard. The current section records the author's attempt at distilling from several such proposals a single set of guidelines for use in private projects. In general, wherever applicable the style guides published online by Google and ID Software have been followed, though with several deviations, most notably with regard to naming conventions for variables.

1.3.1 File structure

At the project level, modularity in C may be achieved through a method advocated by Hanson [7] for separating interfaces from their implementations, growing out of his previous joint work with Fraser [6]. Roughly, given a module, say, `foo`, its interface, as embodied by a header `foo.h`, contains declarations for the types it defines as well as for the operations permitted thereupon.

12a ⟨*foo.h* 12a⟩≡

```
#ifndef FOO_H_

⟨includes 12c⟩
/* defines */
/* typedefs */
/* enums */
/* structs */
/* unions */
/* extern global variable declarations */
/* function prototypes */

#endif /* FOO_H_ */
```

To prevent the header's multiple inclusion in a project, its contents are wrapped inside what are commonly called internal include guards. The corresponding implementation is contained in `foo.c`:

12b ⟨*foo.c* 12b⟩≡

```
#include "foo.h"
⟨includes 12c⟩
/* defines */
/* global variable definitions */
/* static function prototypes */
/* function definitions */
```

As per the Google style guide, we shall want `foo.h` to be self-contained, so guaranteed by insisting on its topmost inclusion in `foo.c`. The order of the remaining includes in both interface and implementation is furthermore constrained as follows:

12c ⟨*includes* 12c⟩≡ (12)

```
/* POSIX headers */
/* C standard library headers */
/* one's own project's headers */
```

In the above, we deviated in one important respect from Hanson. Arguably the cornerstone of his proposal, the separation between interface and implementation allows the former to hide the latter's representation details through use of opaque pointers. Thus, any change in such internals may leave the header untouched, preventing files depending thereon from having to be recompiled. Here, we instead chose to collect *all* `typedef`s, `enum`s, `union`s and `struct`s in headers, mostly to enjoy direct property access, though at the cost of tighter coupling. While we continue to speak of interfaces and their implementations, it should thus be noted that our use of this terminology does not strictly follow Hanson's (if not constituting a case of straight-out blasphemy).

1.3.2 Naming

The following rules apply to the naming of identifiers:

0. Function names are verbs written in `UpperCamelCase`.
1. Typenames, comprising `typedef`s and `struct`-, `enum`- and `union` tags, are noun phrases written in `lowerCamelCase`.
2. Variable names are nouns written in `lower_snake_case`.
3. Compile-time constants, whether part of an `enum` or `#define`d, are written in `UPPER_SNAKE_CASE`, with the exception of function macro's, which are held to the same rules as stipulated for function names.

Names commonly chosen for variables are `it` (for iterators) and `me` (for method parameters of the type exported by an interface, akin to `this` in C++ and Java). While at its core C provides no facilities for organizing one's identifiers into namespaces, we can compensate by augmenting the above rules with the following:

0'. Function names exposed through an interface `foo` are prefixed by `Foo_`.
1'. `typedef`s, `struct`-, `enum`- and `union` tags are postfixed by `_t`, `_s`, `_e` and `_u`, respectively.
2'. Global variables are prefixed by `g_`.
3'. Identifiers defined as part of include guards are postfixed by `_H_`.

In general, we prepare the use of an `enum` over a `#define` for compile-time integral constants.

1.3.3 Whitespace

Each variable declaration is written on its own line, with identation applied in order to align variable names. In particular, `*` and `const` are considered part of the type. E.g.,

13 ⟨*variable declaration example* 13⟩≡

```
const char    foo;
char * const  bar;
```

Module	Interface	Implementation	Section
Circular linked lists	`node.h`	`node.c`	§2.1
Fixed-Size Memory pools	`pool.h`	`pool.c`	§2.3
Exceptions	`except.h`		§2.2
Abstract syntax trees	`ast.h`	`ast.c`	§3.1
Environments	`env.h`	`env.c`	§3.2
Interpreter	`cam.h`	`cam.c`	§3.3
Optimizer	`optim.h`	`optim.c`	§3.4
Lexer	`lexer.h`	`lexer.c`	§4.2
Parser	`parser.h`	`parser.c`	§4.3
Read-Eval-Print Loop		`main.c`	§4.4

Table 1.1: An overview of the files making up our implementation of the CAM.

1.3.4 Braces

The following rules govern the use of curly braces:

0. Braces are mandatory for statement blocks following `if`, `else`, `for`, `do` and `while`, except when empty and following `for` or `while`, in which case a single properly indented occurrence of `;` on a separate line suffices.

1. An opening brace is to be followed by a line break. If following a keyword introducing a control structure, it is to appear on the same line. Otherwise, if following a method declaration, it is to appear immediately preceded by a line break.

2. A closing brace if to appear after a line break, being optionally followed by `else`, or by a line break otherwise.

1.4 Overview

Table 1.1 presents an overview of the project as a whole, naming its modules in their order of presentation, together with their decomposition into interfaces and implementations.

The main design decision that had to be settled prior to commencing work on this project concerned which data structures to use. Wherever possible we preferred simple linked allocation schemes, motivating the start of our discussion with the circular linked lists that lie at the basis of many of our algorithms. Most notably they feature prominently in the implementation of fixed-size memory pools, replacing the C standard library's `malloc` and `free` on account of the observation that we shall need but three different object sizes to be allocated on the heap. Both circular linked lists as well as memory pools are discussed at considerable length by Knuth [11], and our treatments thereof borrow much from his work.

We continue with an account of how λ-terms are represented in-memory. It turns out that while the use of symbolic identifiers for denoting variables aids comprehension by the human mind, less benefit can be claimed when terms so represented are to be manipulated by machine. In particular, such an endeavour would quickly run into problems concerning the conditions under which two terms differing only by a renaming of variable occurrences may be identified. Though questions worthwile of study, as we indeed argued before, they nonetheless primarily constitute artifacts of the choice of representation and need not clutter our computations. Thus, prior to their evaluation, we shall first transform terms into a 'nameless' form better amenable to computation, more precisely replacing its variables with numeric identifiers indicating their distance to the binding λ. First presented by De Bruijn [5], his was not the first proposal for dispensing with bound variable names. Without going into much detail, earlier efforts had concentrated on eliminating abstractions in favour of a number of combinators, essentially higher-order functions, that could be combined using application (see, e.g., Curry et al. [4] for an overview). It was Curien [3] who realized that De Bruijn's representation could itself be understood as a language of combinators, having approached it from the perspective of yet another branch of research, to wit category theory. Often lovingly dubbed 'abstract mathematical nonsense' by its practicioners, the latter's main concern has grown to encompass a paradigmatic shift in mathematical thinking that extends deep into foundational studies. Luckily, our own dealings with category theory in the chapters to come will carry us considerably less far afoot. Curien's combinatorial rephrasing of De Bruijn's proposals may be considered what is often referred to in the computer science literature by an *abstract syntax* for λ-terms, constituting a means of representation free of the peculiarities that go with their more usual *concrete* rendering for the human eye, such as concerning variable renaming. It is, in essence, Curien's proposal that we shall use for our own in-memory representation of terms as well.

Curien's efforts lead directly to the invention of the Categorical Abstract Machine in joint work with Cousineau and Mauny [1], and we shall present its implementation as a traversal over abstract syntax trees. In essence, its workings capture the computational process of *evaluating* a term, in the sense of obtaining from it the value that it denotes. Much like how the interpretation of an utterance depends on context, so in fact the evaluation of a term takes place inside an *environment*. More precisely, by the latter we shall understand a *binding* of the term's free variable occurrences to concrete values, it hopefully being not too far a stretch of the imagination that any choices made in this regard might affect the outcome of determining the value for a term as a whole. It should be noted at this point that Cousineau et al. used a slightly different terminology, referring by terms to what we have here called environments.

Besides *evaluating* a term, we can sometimes also *transform* it without changing the value that it denotes. To illustrate using our concrete syntax, it should be clear that $((\lambda x.(+\ x\ 1))\ 2)$ and $(+\ 2\ 1)$ both evaluate to 3. Though thus far having largely remained silent ont this topic, evaluating and transforming a term may really be considered two sides of the same coin, with much of

the theory of λ-calculus having in fact been devoted to a systematic investigation into the properties of such syntactic rewritings. For our purposes, we may implement them as optimization passes over our abstract syntax trees prior to evaluating them, in so doing following a proposal by Cousineau et al.

Having described a means of representing λ-terms in memory in a form amenable to both their further transformation and ultimate evaluation, the 'backend' of our work is complete. What remains is the 'frontend' task of writing a lexer and parser that transforms a human-readable representation of λ-terms into abstract syntax trees. From the outset we made a conscious decision to keep this part of the work as simple as possible, seeing as our motivation in starting this project largely concerned the CAM. Thus, we shall provide little more than a proof of concept for our implementation of the latter, presenting a restricted subset of the λ-calculus capable, through the addition of appropriate constants, of doing some simple arithmetic. Finally a REPL is thrown in to enable the user to present closed instances of such terms (i.e., without any free variable occurrences), which are then fed to the optimizer and evaluator. This concludes our work.

Chapter 2

Memory management

The current chapter discusses algorithms for managing dynamically allocated resources, lifted largely from Knuth [11]. Overall, we favoured linked representations for our data structures, most of which can always be stored on the stack. What remains are but three distinct object sizes that must be acquirable at runtime, to which effect fixed-size memory pools are particularly well suited given their attractive properties of constant-time allocation and deallocation. We start our exposition in §2.1 with a relatively self-contained implementation of circular linked lists, offering a gentle introduction to our use of literate programming. Next, after a brief interlude about error handling in §2.2, the fruits of our efforts will be used in §2.3 as the basis for implementing memory pools.

2.1 Circular linked lists

In Volume I of TAOCP, Knuth [11] offers a comprehensive account of linked representations for basic data types, including various sorts of both lists and trees. Most of what is to follow in this text builds forth in one way or another on but a small part of Knuth's discussion, although his account of cyclic linked lists shall re-occur time and again as the basis for more involved data structures. Specifically, they can be used for implementing the operations of both a stack and queue in constant time, besides being easily concatenated. Though the use of another link for every node would have provided even more flexibility, we shall find that for our own purposes, singly-linked lists, when circular, hit the sweet spot.

2.1.1 Interface

Linked lists are made up of nodes, and it is by that name to which we shall refer to our module.

17 ⟨*node.h* 17⟩≡

```
#ifndef NODE_H_
```

```
#define NODE_H_

⟨node.h macros 19d⟩
⟨node.h typedefs 18a⟩
⟨node.h structs 18b⟩
⟨node.h function prototypes 19a⟩

#endif /* NODE_H_ */
```

A circular linked list is like an ordinary singly-linked list, except that the last node points to the first. A certain realization may dawn upon a moment's reflection that what comes first and what last is now no longer as clear-cut as it used to be, and we shall resolve the matter by having the client code decide. Specifically, we will write our interface so as to ask for each operation a pointer to what should be deemed the last node of a list, noting the first to then be removed by only a single dereference.[1]

18a ⟨*node.h typedefs* 18a⟩≡ (17)
```
typedef struct node_s node_t;
```

Individual nodes are woven into a list by having each point to its successor, referring to such references by links. What may be initially surprising is that we don't include any fields for holding the data into the definition of a node. Ordinarily void pointers are often used for this purpose, constituting the C language's 'generic' pointer type. This approach proves limiting, however, when basic types must be stored, or when the data is to be spread out over multiple fields. Another approach, used, for instance, in the Linux kernel, relies on the address of a struct coinciding with that of its first field. By restricting the definition of a node to capture only the concept of a link, we can then embed it as the first element of another struct that holds the data fields, with the above property guaranteeing we can freely cast between the two types. Later, we shall see how this technique can also be used to perform object-oriented programming in C. To move our discussion back to cyclic linked lists proper, we note that, compared to their non-cyclic counterparts, we cannot rely on `NULL` to terminate the list, although they will still be used for representing empty ones.

18b ⟨*node.h structs* 18b⟩≡ (17)
```
struct node_s {
  node_t *  link;
};
```

[1] In contrast, had we understood our pointers to identify the first node, then finding the last would take linear time again.

Due to their circular nature, we shall be able to implement the addition of nodes at both the front and back of lists in constant time. The list itself we represent by a double pointer type, as needed for maintaining the invariant that its reference is a pointer to the last node. The node to be added, in turn, may be more conventionally communicated through a single pointer.

19a ⟨*node.h function prototypes* 19a⟩≡ (17) 19b▷
```
extern void     Node_AddFirst(node_t ** const, node_t * const);
extern void     Node_AddLast(node_t ** const, node_t * const);
```

Had we used two links per node, we could similarly have implemented removal from both ends of a list in constant time as well. With but a single link, on the other hand, the best we can do without expanding into linear time is to remove the first node. Though this operation returns a pointer to a node, we instead used a void pointer for typing the result. The reason for this is that in most cases, what is passed to these methods is not just a double pointer to a node by itself, but rather to one that is placed adjacent in memory to its data fields as part of another struct. By returning a void pointer, we can relieve the client somewhat from having to explicitly cast the result back to the proper struct type.

19b ⟨*node.h function prototypes* 19a⟩+≡ (17) ◁19a 19c▷
```
extern void *   Node_RemoveFirst(node_t ** const);
```

Besides expanding a list by a single node, we shall also find the occassional need to concatenate a given list with another at either one of its ends. Both these operations we shall again be able to perform in constant time.

19c ⟨*node.h function prototypes* 19a⟩+≡ (17) ◁19b
```
extern void     Node_Prepend(node_t ** const, node_t * const);
extern void     Node_Append(node_t ** const, node_t * const);
```

Our choice for separating the data fields from the definition of a node proper comes with the consequence of the arguments passed to the methods declared by the current interface having to first be explicitly 'down'-casted to a node. To prevent client code from becoming cluttered with such casts, we export a set of macros that can do the work for us. In naming them, we have made explicit their use in implementing the operations of stacks and queues.

19d ⟨*node.h macros* 19d⟩≡ (17) 20a▷
```
#define Push(me,np)    Node_AddFirst((node_t **)(me),(node_t *)(np))
#define Pop(me)        Node_RemoveFirst((node_t **)(me))
#define Enqueue(me,np) Node_AddLast((node_t **)(me),(node_t *)(np))
#define Append(me,np)  Node_Append((node_t **)(me),(node_t *)(np))
#define Prepend(me,np) Node_Prepend((node_t **)(me),(node_t *)(np))
```

Besides macros wrapping our function declarations, we add another three for finding the successor of a node, testing whether a given list is empty, and, finally, for returning the head of a list, or `NULL` if empty.

20a ⟨*node.h macros* 19d⟩+≡ (17) ◁19d

```
#define Link(me)    (void *)(((node_t *)(me))->link)
#define IsEmpty(me) ((me) == NULL)
#define Peek(me)    (void *)(IsEmpty(me) ? NULL : ((node_t *)(me))->link)
```

2.1.2 Implementation

As noted before, the implementation of circular linked lists that follows is based largely on their discussion by Knuth [11]. The difficulty we shall mostly find to lie in taking into account the cases where one of a method's parameters refers to the empty list, i.e., equals `NULL`.

20b ⟨*node.c* 20b⟩≡

```
#include "node.h"

#include <assert.h>
#include <stddef.h>

⟨node.c function definitions 20c⟩
```

In adding a node to the front of a list, we must consider two cases. If the list is not empty, we insert the new node in between its tail and head, whereas if instead the list is empty, the new node *becomes* the list, pointing to itself.

20c ⟨*node.c function definitions* 20c⟩≡ (20b) 21a▷

```
void
Node_AddFirst(node_t ** const me, node_t * const np)
{
  assert(me);
  assert(np);

  if ((*me)) {
    np->link = (*me)->link;
    (*me)->link = np;
  } else {
    *me = np;
    np->link = np;
  }
}
```

If we can add a node to the front of a list, then we can add it to the back as well simply by additionally updating the list's tail.

21a ⟨*node.c function definitions* 20c⟩+≡ (20b) ◁20c 21b▷

```
void
Node_AddLast(node_t ** const me, node_t * const np)
{
  assert(me);
  assert(np);

  Node_AddFirst(me, np);
  *me = np;
}
```

In removing the head of a list, we have two corner cases to consider. First, the list may be empty, in which case we choose to return `NULL`. Perhaps somewhat less obviously, special care must be taken as well if the removed node was the only one in the list, requiring us to set the latter to `NULL`.

21b ⟨*node.c function definitions* 20c⟩+≡ (20b) ◁21a 22a▷

```
void *
Node_RemoveFirst(node_t ** const me)
{
  node_t *  np;

  assert(me);

  if (IsEmpty(*me)) {
    return NULL;
  }

  np = (*me)->link;
  (*me)->link = np->link;
  if (*me == np) {
    *me = NULL;
  }
  return np;
}
```

Our interface exported two methods for concatenating lists. The difference is one of perspective: between the two input lists, one serves the double purpose of also storing the output, being represented by a double pointer. If the in parameter is added to the front of the in-out parameter, we speak of prepending, otherwise of appending. Starting with our implementation of the former, we must be careful to account for the possibility that either input might refer to an empty list. If neither does, we simply make the tails of both lists point to the other's head.

22a ⟨*node.c function definitions* 20c⟩+≡ (20b) ◁21b 22b▷

```
void
Node_Prepend(node_t ** const me, node_t * const np)
{
  node_t *  tmp;

  if (IsEmpty(np)) {
    return;
  }
  if (IsEmpty(*me)) {
    *me = np;
    return;
  }
  tmp = np->link;
  np->link = (*me)->link;
  (*me)->link = tmp;
}
```

Appending differs little from prepending, except that now the in-out parameter's reference will also have to be updated when pointing at a non-empty list.

22b ⟨*node.c function definitions* 20c⟩+≡ (20b) ◁22a

```
void
Node_Append(node_t ** const me, node_t * const np)
{
  node_t  *tmp;

  if (IsEmpty(np)) {
    return;
  }
  if ((*me)) {
    tmp = np->link;
    np->link = (*me)->link;
    (*me)->link = tmp;
  }
  *me = np;
}
```

2.2 Error handling

Hanson [7] distinguishes three types of errors. The most common are user errors, like specifying a non-existing file. The caller is typically notified via a status code, or through the return of a special value. Error recovery, e.g., for the purpose of cleaning up acquired resources, be it within the method where the error occurred or its caller, is usually separated from the main program flow through a local forward jump. Next are errors in the programming itself, i.e., bugs, from which recovery is impossible. At least for test builds, such circumstances may be caught using the standard library's `assert` facilities. Third, and finally, are exceptions. Like with user errors, recovery may sometimes be possible, though exceptions usually occur far less frequently. As typical examples Hanson mentions various cases of under- and overflow, as well as, what currently interests us the most, memory depletion. Because of their infrequent occurrence, exceptions are ill-served by status codes, cluttering up the normal program flow with code for rare conditions. Instead, the C standard library offers `setjmp` and `longjmp` for enabling non-local jumps, allowing code for handling the exception to be completely separated from program logic distributed over multiple methods. In general, however, exceptions may occur nested, sometimes even having to be re-raised within a handler in order for another handler higher in the chain to process it. Besides that, a given handler sometimes has to distinguish between multiple types of exceptions, applying a different recovery logic to each. `setjmp` and `longjmp` are too primitive to address all these use cases conveniently, and for this purpose are often taken as the basis for writing higher-level abstractions with. Our own situation is less complex: we shall not have to differentiate between exceptions based on their type, nor do we require more than a single handler. On the hander hand, we *will* have to be able to raise exceptions on more occassions than for handling memory errors alone, so that at least some degree of abstraction is warranted, though of far less generality than is needed for supporting every use case imaginable.

23 ⟨*except.h* 23⟩≡

```
#ifndef EXCEPT_H_

#include <setjmp.h>
#include <stdlib.h>

⟨except.h macros 24b⟩
⟨except.h variable declarations 24a⟩

#endif /* EXCEPT_H_ */
```

To enact a non-local jump, the jump site must be known first. To this end, the C standard library offers the method `setjmp`, to be called prior to `longjmp` for executing the actual jump. Since nothing prevents us from invoking the former more than once, we need a means of synchronization to match `longjmp` to the right call of `setjmp`. Such is achieved by having both take as argument an object of type `jmp_buf`. For our own purposes, we shall require but a single handler for processing all exceptions, so that only one `jmp_buf` object will suffice. Still, we shall want to make sure that `setjmp` has in fact been called thereon before invoking `longjmp`. By exporting a *pointer* to a `jmp_buf` object, we can by default set it to `NULL` and initialize it only prior to the call to `setjmp`, making a null-check suffice before invoking `longjmp` to affirm that the jump site has indeed been set.

24a ⟨*except.h variable declarations* 24a⟩≡ (23)

```
extern jmp_buf *g_handler;
```

The way `setjmp` works is that it returns twice: the first time with 0, and afterwards with a non-zero value to indicate an exception was raised. It is thus conventionally used as the condition of an `if` statement (or sometimes as part of a `switch`, if multiple exception type have to be distinguished), with the normal- and exceptional program flows being coded as the then- and else clauses. We will wrap this idiom inside a syntax akin to that of Java's try/catch mechanism using the macro's below, allowing us to render it by `TRY <normal flow> CATCH <exception flow> END`. Note that in doing so, we have to be careful to set `g_handler` to non-`NULL` prior to the `TRY` clause, and to reset it back to `NULL` after the `CATCH`.

24b ⟨*except.h macros* 24b⟩≡ (23) 25▷

```
#define TRY {                       \
  jmp_buf handler;                  \
                                    \
  g_handler = &handler;             \
  if (!setjmp(handler)) {

#define CATCH } else {

#define END }                       \
    g_handler = NULL;               \
  }
```

To `THROW` an exception, we must first make sure that the jump site has been set by validating `g_handler` is non-`NULL`, otherwise simply exiting. Note furthermore that `longjmp` takes a second integral argument, indicating the value returned by `setjmp`. If multiple exception types are to be distinguished, here is the place to do it. For our purposes, however, we can simply always return 1. Finally, to allow the user to write `THROW;`, i.e., including the semi-colon, we have applied a standard trick by wrapping our macro inside a do-while.

25 ⟨*except.h macros* 24b⟩+≡ (23) ◁24b

```
#define THROW do {                \
  if (*g_handler) {               \
      longjmp(*g_handler, 1);     \
  } else {                        \
    exit(1);                      \
  }                               \
} while (0)
```

2.3 Fixed-size memory pools

The C standard library implements a general-purpose memory management scheme capable of allocating objects of any size. By furthermore requiring each invocation of `malloc` or `calloc` to have a corresponding call to `free`, object lifetimes are similarly unconstrained as well. Though suitable for any use case, this generality comes at a performance cost. Besides being prone to fragmentation, allocations require a linear search through a linked list of memory blocks for free space large enough to serve the request. These shortcomings become particularly apparent when a large number of relatively small short-lived objects must be created, happening to be the primary use case for the program under consideration. Various alternative solutions to dynamic memory management have been explored, however, that sacrifice generality in exchange for performance. For our own purposes, it turns out that a simple solution described by Knuth [11] suffices. Specifically, since we shall find occasion to allocate objects of but three different sizes on the heap, we can do so using separate dedicated memory pools. Each serves requests for only a single size, eliminating fragmentation entirely and making it possible to implement constant time allocation. Object lifetimes, on the other hand, we shall want to keep unconstrained, thus still requiring each object to be freed individually. This contrasts with, e.g., the concept of an arena used by Fraser and Hanson [6], [7], whereby deallocations are performed in batches. Such is possible only whenever one's objects form natural groupings in terms of their lifetimes, however, which did not consistently apply in our own case.

2.3.1 Interface

A memory pool enables constant-time allocation and deallocation for objects of a fixed size.

26 ⟨*pool.h* 26⟩≡

```
#ifndef POOL_H_
#define POOL_H_

#include <stddef.h>

#include "node.h"

⟨pool.h macros 28a⟩
⟨pool.h constants 28b⟩
⟨pool.h typedefs 27a⟩
⟨pool.h variable declarations 27d⟩
⟨pool.h function prototypes 28c⟩

#endif /* POOL_H_ */
```

The size of the objects served by a given memory pool is recorded as part of its definition. The C standard library exports an unsigned integral type `size_t` large enough to hold the size of any object, being perfectly suited to our purposes.

27a ⟨*pool.h typedefs* 27a⟩≡ (26)
```
typedef struct {
  size_t        size;
  ⟨pool_t fields 27b⟩
} pool_t;
```

A memory pool allocates its objects from a byte array that we shall set aside for it at compile time, offering a fixed capacity that is a multiple of `size`. The fields `start` and `limit` delimit the array, pointing at its first byte, resp. one beyond the last. Finally, `max` points at the next allocable byte, its distance from `start` again always being a multiple of `size`.

27b ⟨*pool_t fields* 27b⟩≡ (27a) 27c▷
```
char * const  start;
char * const  limit;
char *        max;
```

If the lifetimes of all our objects always adhered to last-in first-out, we would be done: `max` could be incremented for every allocation, and decremented again upon a deallocation. Rarely is memory management so simple, however, and our situation is no exception. To enable deallocation without restrictions, we will maintain a separate linked list of available free objects. Releasing an object will simply put it at the head, whereas allocation will first try to take an object from the list before resorting to incrementing `max`. The free list itself may be typed simply using `node_t`, as all objects whose allocation we shall want to manage using a pool will extend therefrom.

27c ⟨*pool_t fields* 27b⟩+≡ (27a) ◁27b
```
node_t *      avail;
```

We will need a total of three memory pools for serving allocation requests, which we shall globally declare here together. Our reasons for exporting their identities like so are primarily due to enable proper exception handling. Specifically, if we end up running out of memory in one pool, we shall recover by releasing all resources held by every pool before allowing the user to try inputting another term through the REPL.

27d ⟨*pool.h variable declarations* 27d⟩≡ (26)
```
extern pool_t   g_ast_pool;
extern pool_t   g_env_pool;
extern pool_t   g_symbol_pool;
```

Pools are always initialized the same way, suggesting the use of a macro. We require the type of objects being served, allowing to deduce their size, as well as a reference to a backing array together with its capacity.

28a ⟨*pool.h macros* 28a⟩≡ (26) 28d▷
```
#define INIT_POOL(pool, elems, type) {              \
  sizeof(type),                         /* size */    \
  (char * const)(pool),                 /* start */   \
  (char * const)((pool) + (elems)),     /* limit */   \
  (char *)(pool),                       /* max */     \
  NULL                                  /* avail */   \
}
```

In practice, we shall use the same number of elements for every array backing a memory pool, defining it here once by a constant.

28b ⟨*pool.h constants* 28b⟩≡ (26)
```
enum {
  N_ELEMS = 1024
};
```

Similarly to the standard library, two methods are exported for memory allocation. The objects returned by the first, `Pool_Alloc`, may contain garbage, whereas those returned by `Pool_Calloc` have all their bits set to 0. Note that in both cases we return a void pointer, enabling implicit casts to any type of struct extending `node_t`.

28c ⟨*pool.h function prototypes* 28c⟩≡ (26) 28e▷
```
extern void *   Pool_Alloc(pool_t * const);
extern void *   Pool_Calloc(pool_t * const);
```

Allocated objects may be individually released using `Pool_Free`, similarly to the standard library's `free`. In addition, we also allow for entire lists to be freed at once using `Pool_FreeList`.

28d ⟨*pool.h macros* 28a⟩+≡ (26) ◁28a
```
#define Pool_Free(me, item)       Push(&(me)->avail, (item))
#define Pool_FreeList(me, item)   Append(&(me)->avail, (item))
```

Finally, a memory pool can be cleared in the sense of having all its memory released. This invalidates all objects previously allocated from it that had not yet been freed, placing the responsibility with the client to no longer refer to them. In practice, we shall only use this method for recovering from out of memory exceptions.

28e ⟨*pool.h function prototypes* 28c⟩+≡ (26) ◁28c
```
extern void     Pool_Clear(pool_t * const);
```

2.3.2 Implementation

As with cyclic linked lists, so our implementation of memory pools is based largely on Knuth [11].

29a ⟨*pool.c* 29a⟩≡

```
#include "pool.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "except.h"
#include "node.h"

⟨pool.c function definitions 29b⟩
```

We first try to satisfy allocation requests from the list of available freed objects. Only if the latter is empty do we attempt to retrieve the required space from the backing array by incrementing `max`. Finally, if that fails as well, we print a message and raise an exception.

29b ⟨*pool.c function definitions* 29b⟩≡ (29a) 30a▷

```
void *
Pool_Alloc(pool_t * const me)
{
  assert(me);

  if ((me->avail)) {
    return Pop(&me->avail);
  }
  if (me->max < me->limit) {
    me->max += me->size;
    assert(me->max <= me->limit);
    return me->max - me->size;
  }
  fprintf(stderr, "Out of memory.\n");
  THROW;
}
```

`Pool_Calloc` works the same as `Pool_Alloc`, except that it will always clear all bits of an object before returning it to the caller.

30a ⟨*pool.c function definitions* 29b⟩+≡ (29a) ◁29b 30b▷

```
void *
Pool_Calloc(pool_t * const me)
{
  void *  ptr;

  assert(me);

  ptr = Pool_Alloc(me);
  memset(ptr, 0, me->size);
  return ptr;
}
```

Releasing all memory held by a memory pool is as easy as emptying the list of freed objects, and resetting `max` to the start of the backing array.

30b ⟨*pool.c function definitions* 29b⟩+≡ (29a) ◁30a

```
void
Pool_Clear(pool_t * const me)
{
  assert(me);

  me->avail = NULL;
  me->max = (char *)me->start;
}
```

Chapter 3

Evaluating λ-terms

Having laid the groundwork for building linked data structures and managing their lifecycle, we continue with the most data-heavy part of our exposition. Specifically, we will start in §3.1 with a discussion of our in-memory representation for λ-terms, allowing us to abstract away from the peculiarities surrounding named variables. In the subsequent two sections §3.2 and §3.3, we discuss environments, resp. the evaluation of a term relative to a given environment. §3.4 concludes with a number of optimizations that may be applied to a term prior to its evaluation.

3.1 Abstract Syntax Trees

Whereas the previous chapter limited its discussion of data structures to the one dimensional by sticking to linked lists, in the current we will expand into two with the definition of trees. As is becoming a recurring theme by now, we base ourselves largely on the work of Knuth [11], specifically his representation of forests by binary trees.

3.1.1 Interface

The AST module (for Abstract Syntax Tree) exports declarations both for a tree-based in-memory representation of λ-terms (named, unsurprisingly, AST's), as well as for a means to specify what logic to apply during the visits of individual nodes as part of a tree traversal. The latter device will in particular be used later for implementing the evaluation and optimization of terms.

31 ⟨*ast.h* 31⟩≡

```
#ifndef AST_H_
#define AST_H_

#include <stdarg.h>
#include <stdbool.h>
```

```
#include "node.h"

⟨ast.h macros 35c⟩
⟨ast.h typedefs 32a⟩
⟨ast.h structs 32b⟩
⟨ast.h function prototypes 35b⟩

#endif /* AST_H_ */
```

The nodes of an abstract syntax tree may contain any arbitrary number of children. This complicates their definition, compared to, say, that of a binary tree, although the increased generality will make our lifes easier in other respects later on. To be clear, we use the word 'node' in what is to follow to refer to abstract syntax trees wherever confusion does not arise with its previous usage for cyclic linked lists. That said, we shall see presently that a node in the current sense may be regarded as a specialization of a list.

32a ⟨*ast.h typedefs* 32a⟩≡ (31) 33a▷

```
typedef struct ast_s ast_t;
```

The children of a node are accessed through a pointer `rchild` to its rightmost one, heading a cyclic linked list. We make the list structure part of the definition of a node by having it extend `node_t`, whose `link` field here thus acts as a pointer to a node's right sibling.

32b ⟨*ast.h structs* 32b⟩≡ (31) 38b▷

```
struct ast_s {
  node_t    base;
  ast_t *   rchild;
  ⟨ast_s fields 32c⟩
};
```

We distinguish between several types of nodes, each corresponding (for the most part) to a construct of the source language, i.e., λ-terms in the usual named notation. In one case, this happens to be an integral constant, requiring storage in a field of its own.

32c ⟨*ast_s fields* 32c⟩≡ (32b)

```
int       value;
astType_t type;
```

When evaluating a term, its free variables represent unknowns. We thus have no choice but to parameterize over their denotations if we are to come up with a value for the term as a whole. We can make these remarks more explicit by introducing the concept of a binding environment. Though we will flesh out its definition over the coming paragraphs, for now it suffices to consider it as representing a choice of denotations for a term's free variables. A term itself we may then represent by a function over such a binding environment, s.t. the computation thereof for some argument corresponds to the term's evaluation. An AST, in turn, is simply an algebraic description of such a function. Its leafs, in particular, correspond to functions that we may treat as 'primitive' for our purposes, whereas intermediate nodes constitute operators acting on existing functions (i.e., as represented by its children) for deriving new ones. The *type* of a node tells us which primitive function or operator it denotes.

33a ⟨*ast.h typedefs* 32a⟩+≡ (31) ◁32a 36d▷

```
typedef enum {
  ⟨leaf node types 33c⟩
  ⟨parent node types 33b⟩
} astType_t;
```

One way in which to combine functions is by composing them. Specifically, if f, g are functions s.t. f's codomain is a subset of g's domain, then we define their composition $(g \circ f)$ by $(g \circ f)(x) = g(f(x))$. One way to understand this is as a *substitution* of $f(x)$ for x in g. Function composition is associative, and we will represent $(f_n \circ \ldots \circ f_1)$ by a node of type `AST_COMP` whose children, from left to right, represent $f_1, \ldots, f_n$.

33b ⟨*parent node types* 33b⟩≡ (33a) 33d▷

```
AST_COMP,
```

Viewed algebraically as a binary operation on functions, composition has an identity 'element' Id s.t. $Id(x) = x$, represented by a leaf node of type `AST_ID`, meaning $(f \circ Id) = (Id \circ f) = f$.

33c ⟨*leaf node types* 33c⟩≡ (33a) 34a▷

```
AST_ID,
```

Besides composing then, functions can also be *paired*. Specifically, if f, g are functions with the same domain, $\langle f, g \rangle$ is defined by $\langle f, g \rangle(x) = (f(x), g(x))$, where (u, v) is an ordered pair of elements u, v in the set-theoretic sense.

33d ⟨*parent node types* 33b⟩+≡ (33a) ◁33b 35a▷

```
AST_PAIR,
```

Using pairing and composition, we can define the representation for an application $(s\ t)$. If the terms s, t are represented by functions s', t' resp., then $\langle s', t' \rangle$ enables independent evaluation of s and t w.r.t. any binding environment Γ, recording the results in an ordered pair $(s'(\Gamma), t'(\Gamma))$. If we define a function *App*, encoded by a leaf node of type `AST_APP`, s.t. $App(u, v) = u(v)$, we can pass it our intermediate result $(s'(\Gamma), t'(\Gamma))$ to obtain the desired function application $s'(\Gamma)(t'(\Gamma))$ for evaluating $(s\ t)$ within the binding environment Γ.

34a ⟨*leaf node types* 33c⟩+≡ (33a) ◁33c 34b▷
```
AST_APP,
```

Though the first chapter presented the λ-calculus as a means of unifying a wide range of notations used across different subfields of mathematics, our ambitions as to its current implementation are less far-reaching. We shall in particular stick to the non-negative integers n for our domain of discourse, which we may each represent by a *constant* function $'n$, i.e., s.t. always $('n)(\Gamma) = n$. The mapping taking an integer to its corresponding constant function is referred to by Cousineau et al. [1] as *Quoting*.

34b ⟨*leaf node types* 33c⟩+≡ (33a) ◁34a 34c▷
```
AST_QUOTE,
```

To narrow down the scope of this work even further to little more than a proof of concept, we shall consider but a single operation from arithmetic, specifically addition. If so desired, the reader should find little challenge in building forth on our current efforts to enable support for other operations as well.

34c ⟨*leaf node types* 33c⟩+≡ (33a) ◁34b 34d▷
```
AST_PLUS,
```

So far, we had little to say about what defines a binding environment, instead contending ourselves with a sense of its intended purpose. On first approximation, we may think of it as a mapping from variable names to values. We can do better, however, by keeping only the values and replacing variable names in a term with addresses , serving as instructions for looking up a denotation inside an environment. Specifically, let an environment be a tuple $(v_1, \ldots, v_n)$ of values $v_1, \ldots, v_n$ for arbitrary n. A variable may then be represented by a *projection* function π_i for some $1 \leq i \leq n$, s.t. $\pi_i(v_1, \ldots, v_n) = v_i$. In practice, we shall admit only ordered pairs instead of arbitrary tuples, so that $(v_1, \ldots, v_n)$ is to be understood as abbreviating $(((v_1, v_2), \ldots), v_n)$. A projection π_i, in turn, is decomposed as $Snd \circ Fst^{n-1}$, where *Fst* and *Snd* are the first- and second projections respectively, and Fst^{n-1} means the composition of $n-1$ times *Fst*.

34d ⟨*leaf node types* 33c⟩+≡ (33a) ◁34c
```
AST_FST,
AST_SND,
```

Though the notion of environment was specialized to an ordered tuple of values, we have not been precise as to what a value ís, other than treating it intuitively as something that may result from evaluating a term. We will in fact require only two types of values, the first obviously being non-negative integers. The second, in turn, will reveal itself upon a closer examination of the in-memory representations for abstractions. What to make of $(\lambda x.t)$ if the AST for t has replaced all names with addresses (i.e., compositions $Fst^{n-1} \circ Snd$)? The question, of course, is what address was substituted for x? If we systematically encode bound variables u by projections π_i for i the number of λ's seen when traveling up the parse tree until λu is first encountered, then it follows that all free occurrences of x in t must have been replaced with *Snd*. As such, $(\lambda x.t)$ becomes a mapping from environments Γ to that function which, when applied to a value v, returns $t'(\Gamma, v)$, for t' the AST of t. But this is simply the Curried form of t', which we will write $\Lambda(t')$. Now notice that, contrary to the value of, e.g., an integral constant, the result of evaluating $(\lambda x.t)$ in an environment Γ is not a number but instead a *function* $\Lambda(t')(\Gamma)$. This, then, is the second type of value that we shall recognize, referring to it by a *closure*.

35a ⟨*parent node types* 33b⟩+≡ (33a) ◁33d
```
AST_CUR
```

Knowing now what AST's look like, we move on to instantiating them. The Standard Library offers facilities for writing variadic functions, enabling us to pass in (in left-to-right order) an arbitrary number of children. In addition, we shall need to know the node type, as well as how many children there are.

35b ⟨*ast.h function prototypes* 35b⟩≡ (31) 36a▷
```
extern ast_t * Ast_New(const astType_t, int, ...);
```

The above method puts no restrictions on the node type or on the number of children that are passed in. We can specialize its application for particular types using the definitions below, noting the use of a variadic macro in case of compositions.

35c ⟨*ast.h macros* 35c⟩≡ (31) 36b▷
```
#define Ast_Id()                Ast_New(AST_ID, 0)
#define Ast_Fst()               Ast_New(AST_FST, 0)
#define Ast_Snd()               Ast_New(AST_SND, 0)
#define Ast_App()               Ast_New(AST_APP, 0)
#define Ast_Cur(child)          Ast_New(AST_CUR, 1, (child))
#define Ast_Pair(left, right)   Ast_New(AST_PAIR,2,(left),(right))
#define Ast_Comp(cnt, ...)      Ast_New(AST_COMP,(cnt),__VA_ARGS__)
```

The above macros do not yet create instances for nodes of types `AST_QUOTE` or `AST_PLUS`. These, instead, we will create by means of specialized methods. In case of `AST_QUOTE`, we parameterize over an integer, whereas for reasons to be described later, functional constants like addition are represented by compound AST's as opposed to a single node of the required type (i.e., here `AST_PLUS`).

36a ⟨*ast.h function prototypes* 35b⟩+≡ (31) ◁35b 36c▷
```
extern ast_t * Ast_Quote(const int);
extern ast_t * Ast_Plus(void);
```

Sometimes, we may not know in advance which and/or how many children to add to a node. For these situations, we offer macros to create a node initially without children, and to either add these later one by one from right to left, or to set them all at once using a pre-built list.

36b ⟨*ast.h macros* 35c⟩+≡ (31) ◁35c
```
#define Ast_Node(type)                Ast_New((type), 0)
#define Ast_AddChild(me,child)        Push(&(me)->rchild,(child))
#define Ast_SetChildren(me,children) (me)->rchild = (children)
```

Rather than leaving it to the client to clean up an AST one node at a time, we will instead export a method for releasing an entire tree all at once. By taking an argument of type `ast_t **`, we can reset the client's reference to `NULL`, preventing dangling pointers. It should be noted that if the node referred to by the argument has itself any siblings, then these will not be touched.

36c ⟨*ast.h function prototypes* 35b⟩+≡ (31) ◁36a 36e▷
```
extern void    Ast_Free(ast_t ** const);
```

We will use tree walks both for evaluating terms as well as to optimize them in advance. As such, we want to define the logic for traversing an AST only once, abstracting over the actions that are to be applied upon visiting each type of node. This separation of a composite data structure from the operations that may be performed thereon is precisely achieved by the Visitor Pattern, familiar from Object-Oriented languages and described, e.g., by Johnson et al. [9]. Over the next paragraphs we will explain the concept of a visitor and walk through its realization in C step by step, starting from the following definition.

36d ⟨*ast.h typedefs* 32a⟩+≡ (31) ◁33a 37a▷
```
typedef struct visit_s visit_t;
```

We shall now require a tree traversal to parameterize both over the root node from which to commence, as well as over a visitor, containing the actions to apply to the individual nodes encountered along the way.

36e ⟨*ast.h function prototypes* 35b⟩+≡ (31) ◁36c 38c▷
```
extern void    Ast_Traverse(const ast_t * const, visit_t * const);
```

Before proceeding, we will first define a means for our custom actions to provide the tree walker that applies them with feedback, telling it how to proceed through the return of a status code. In most cases we can simply let it continue, although on occasion we shall want it to skip the children of the node that was last visited.

37a ⟨*ast.h typedefs* 32a⟩+≡ (31) ◁36d 37b▷

```
typedef enum {
  SC_CONTINUE,
  SC_SKIP,
} statusCode_t;
```

An action we shall represent by a function pointer. That way, by changing its reference, we change the algorithm, achieving our desired separation. In a way to be made precise below, a visitor we may then consider a choice for a collection of such function pointers, one for each node type. As for the method signature of an action, it should come as no surprise to require at least for a reference to a concrete node to be passed in. In addition, we may want to keep track of state in between individual visits, storing it on the visitor itself alongside its actions. It follows that, conversely, an individual action should know about the visitor containing it, enabling it to access any required state.

37b ⟨*ast.h typedefs* 32a⟩+≡ (31) ◁37a 37c▷

```
typedef statusCode_t (*visitFunc_t)(visit_t * const, const ast_t *);
```

We have alluded to the multiplicity of visitors, each representing a separate algorithm born from a different choice of actions, and possibly containing private state. We are describing polymorphism, i.e., the ability to refer to different types by a single shared interface. The latter, in this case, consists of simply the declarations for actions corresponding to visits of different node types during a tree traversal. While we could make these part of the definition for `visit_s`, doing so would consume space for every single instance, even when some share the exact same choice of actions. To accommodate such sharing, we will instead separate them into a *virtual function table*, as is a standard technique for realizing Object-Oriented programming in C.

37c ⟨*ast.h typedefs* 32a⟩+≡ (31) ◁37b

```
typedef struct {
  ⟨virtual functions 38a⟩
} visitVtbl_t;
```

During a tree traversal, every leaf is visited exactly once. Parent nodes, on the other hand, are seen more often, each time providing an opportunity for applying some action. In particular, we may visit them both prior as well as after having visited their children, speaking, resp., of pre- and postvisiting. In addition, a binary node may also be visited in between visiting its two children. If we declare an action for every opportunity for visiting any of our node types, we end up with the following list.

38a ⟨*virtual functions* 38a⟩≡ (37c)

```
visitFunc_t   VisitId;
visitFunc_t   VisitApp;
visitFunc_t   VisitQuote;
visitFunc_t   VisitPlus;
visitFunc_t   VisitFst;
visitFunc_t   VisitSnd;
visitFunc_t   PreVisitComp;
visitFunc_t   PreVisitPair;
visitFunc_t   PreVisitCur;
visitFunc_t   InVisitPair;
visitFunc_t   PostVisitComp;
visitFunc_t   PostVisitPair;
visitFunc_t   PostVisitCur;
```

We can now define a visitor simply by a pointer to a virtual function table. Specializations can be obtained in the same way that we did for the nodes of a circular linked list. I.e., seeing as a struct's address in C coincides with that of its first field, making the latter a `visit_t` allows for us to downcast the struct as a whole thereto.

38b ⟨*ast.h structs* 32b⟩+≡ (31) ◁32b

```
struct visit_s {
  const visitVtbl_t * vptr;
};
```

In practice, not every algorithm that we wish to express for an AST will utilize each visitor method. We therefore define a 'default' implementation that simply does nothing, allowing us to initialize function pointers for unused actions therewith.

38c ⟨*ast.h function prototypes* 35b⟩+≡ (31) ◁36e

```
extern statusCode_t VisitDefault(visit_t * const, const ast_t *);
```

3.1.2 Implementation

39a ⟨*ast.c* 39a⟩≡
```
#include "ast.h"

#include <assert.h>

#include "pool.h"

⟨ast.c global variables 39b⟩
⟨ast.c function definitions 39c⟩
```

The nodes of an AST have to be dynamically allocated, to which end we define a dedicated memory pool. By making the sizes of their backing arrays known at compile time, pools themselves are always of fixed capacity. More sophisticated implementations are possible where a pool can be dynamically grown as needed, although for the modest purposes of the program under consideration, the benefits of such an endeavour did not seem to outweigh the simplicity of the approach adopted currently.

39b ⟨*ast.c global variables* 39b⟩≡ (39a)
```
static ast_t  g_pool[N_ELEMS];
pool_t        g_ast_pool = INIT_POOL(g_pool, N_ELEMS, ast_t);
```

To create a new node, we specify both its type and its children. The latter can be of arbitrary number, passed in as a separate argument.

39c ⟨*ast.c function definitions* 39c⟩≡ (39a) 40c▷
```
ast_t *
Ast_New(const astType_t type, int cnt, ...)
{
  ast_t * me;
  va_list argp;

  ⟨allocate a new node me of the given type 40a⟩
  ⟨add cnt child nodes 40b⟩
  return me;
}
```

Recall that allocation from a memory pool first attempts to recycle previously freed nodes. The latter form a linked list, references to which may hence be retained as garbage in the returned object. By invoking `Pool_Calloc` instead of `Pool_Alloc`, we make sure all the bits are cleared.

40a ⟨*allocate a new node* `me` *of the given* `type` 40a⟩≡ (39c)
```
me = Pool_Calloc(&g_ast_pool);
me->type = type;
```

We next count down from `cnt` in adding the new node's children, enqueuing them one by one to make sure that their order is retained.

40b ⟨*add* `cnt` *child nodes* 40b⟩≡ (39c)
```
va_start(argp, cnt);
while (cnt-- > 0) {
  Enqueue(&me->rchild, va_arg(argp, ast_t *));
}
va_end(argp);
```

To create a new node of type `AST_QUOTE`, we can proceed as above without adding any children, though now additionally having to specify a numeric value.

40c ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁39c 40d▷
```
ast_t *
Ast_Quote(const int value)
{
  ast_t * me;

  me = Pool_Calloc(&g_ast_pool);
  me->type = AST_QUOTE;
  me->value = value;
  return me;
}
```

We can make the arguments of a functional constant like `+` explicit by abstracting over them, resulting in an AST $\Lambda(+ \circ \mathit{Snd})$. In particular, note $\Lambda(+ \circ \mathit{Snd})(\Gamma)(m, n)$ evaluates to $+(m, n)$ for any environment Γ and numbers m, n, as desired. Using this encoding will simplify the evaluation of applications later in allowing us to assume the operator to always be an abstraction, as opposed to having to make exceptions for functional constants.

40d ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁40c 41a▷
```
ast_t *
Ast_Plus(void)
{
  return Ast_Cur(Ast_Comp(2, Ast_Snd(), Ast_Node(AST_PLUS)));
}
```

To release the resources held by an AST, we first flatten it into a linked list that we can then deallocate all at once, as opposed to freeing each node individually. We can do so by starting with a list containing only the root node, and iterate through while at each step growing it at the end until all nodes have been seen.

41a ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁40d 41c▷

```
static node_t *
Flatten(ast_t *me)
{
  ast_t * it = me;

  assert(me);

  me->base.link = (node_t *)me;
  do {
    Append(&me, it->rchild);
  } while ⟨there are more nodes to process 41b⟩;

  return (node_t *)me;
}
```

At all times, the original root node oocupies the head of our list. Indeed, when we started out it was the ónly element, and hence both the first and the last, while every iteration at most adds nodes to the end. By a similar reasoning, we can ascertain `me` always points to the final element of the list. As such, we know we are done when advancing `it` gives us back `Link(me)`.

41b ⟨*there are more nodes to process* 41b⟩≡ (41a)

```
((it = Link(it)) != Link(me))
```

We can now release an AST by flattening it into a list and deallocating the latter in its entirety. In doing so, we have to make sure not to touch the root node's siblings.

41c ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁41a 42a▷

```
void
Ast_Free(ast_t ** const me)
{
  assert(me);

  if (*me == NULL) {
    return;
  }
  Pool_FreeList(&g_ast_pool, Flatten(*me));
  *me = NULL;
}
```

We move on to tree traversal, essentially consisting of a sequence of visited nodes. The actions to apply thereat we will invoke through an offset into a virtual function table, allowing us to take some shortcuts later on.

42a ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁41c 42b▷

```
static inline statusCode_t
Visit(const ast_t * const me, visit_t * const vp, const size_t offset)
{
  return (*((visitFunc_t *)vp->vptr)[offset])(vp, me);
}
```

Traditionally, one distinguishes between preorder-, inorder- and postorder tree traversals, depending on whether a node is visited prior, in between, or after traversing its children. Considering the difference a property of whichever algorithm is being applied, we have exposed callback methods for each of these opportunities on the visitor interface, leaving it to the implementor to decide which to associate with a concrete action.

42b ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁42a 43c▷

```
void
Ast_Traverse(const ast_t * const me, visit_t * const vp)
{
  ast_t *       ap;
  statusCode_t  sc;

  assert(me);
  assert(vp);

  ⟨previsit 42c⟩
  ⟨traverse children 43a⟩
  ⟨postvisit 43b⟩
}
```

Given a node, we can switch on its type to decide which visitor method to call. However, by having carefully arranged the declaration orders both of the enum constants defining AST types, as well as of the visitor methods, we can more quickly use the former as an offset into a virtual function table to find the function pointer for performing the corresponding (pre)visit.

42c ⟨*previsit* 42c⟩≡ (42b)

```
sc = Visit(me, vp, me->type);
```

Provided a node's previsit did not return `SC_SKIP`, we next recursively traverse its children, if any. A minor complication arises if we are dealing with a pair, in which case we have to call `InVisitPair` after having walked its first child.

43a ⟨*traverse children* 43a⟩≡ (42b)
```
if (sc == SC_CONTINUE && (me->rchild)) {
  ap = Link(me->rchild);
  Ast_Traverse(ap, vp);
  if (me->type == AST_PAIR) {
    Visit(me, vp, 9); /* InVisitPair */
  }
  while ((ap = Link(ap)) != Link(me->rchild)) {
    Ast_Traverse(ap, vp);
  }
}
```

For a node's postvisit, we can pull a trick similar to that applied for its previsit, using the node type for computing an index into a virtual function table.

43b ⟨*postvisit* 43b⟩≡ (42b)
```
if (me->type == AST_CUR || me->type == AST_COMP
    || me->type == AST_PAIR) {
  Visit(me, vp, me->type + 4);
}
```

Not every one of a visitor's methods may be meaningful to a particular implementation. In these cases, we can use the default 'action' of doing nothing. Note this leaves the method parameters unused, and we have inserted vacuous casts to `void` in order to silence compiler warnings.

43c ⟨*ast.c function definitions* 39c⟩+≡ (39a) ◁42b
```
statusCode_t
VisitDefault(visit_t * const me, const ast_t *ap)
{
  (void)me;
  (void)ap;
  return SC_CONTINUE;
}
```

3.2 Environments

The previous section spoke intuitively of binding environments in motivating our choice of AST. The current will make their definition explicit, allowing us in the next to implement evaluation of terms.

3.2.1 Interface

44a ⟨*env.h* 44a⟩≡

```
#ifndef ENV_H_
#define ENV_H_

#include "ast.h"

⟨env.h macros 45e⟩
⟨env.h typedefs 44b⟩
⟨env.h structs 44c⟩
⟨env.h function prototypes 45d⟩

#endif /* ENV_H_ */
```

A (binding) environment is simply a tuple of values that may serve as denotations for the free variables in a term.

44b ⟨*env.h typedefs* 44b⟩≡ (44a) 45a▷

```
typedef struct env_s env_t;
```

Like an AST, an environment is a tree built up from nodes, specializing cyclic linked lists. We distinguish between several types, each with different data fields that we collect together in a union. Similarly to the previous section, we will, until further notice, and so long as no confusion arises, speak of nodes to refer exclusively to the components of an environment.

44c ⟨*env.h structs* 44c⟩≡ (44a)

```
struct env_s {
  node_t        base;
  union {
    ⟨env_s union fields 45c⟩
  }             u;
  envType_t     type;
};
```

Values are carried at the leafs and can be either non-zero integers or closures. Using pairing, we combine them together into larger structures. Finally, we reserve a 'sentinel' node type for 0-tuples, used to evaluate closed terms in.

45a ⟨*env.h typedefs* 44b⟩+≡ (44a) ◁44b 45b▷
```
typedef enum {
  ENV_PAIR,
  ENV_NIL,      /* sentinel */
  ENV_INT,      /* non-zero integers */
  ENV_CLOSURE,
} envType_t;
```

We previously spoke intuitively of a closure as the value of an abstraction. Specifically, we may say a closure consists of the environment Γ wherein an abstraction was evaluated, together with the latter's body, as represented, say, by some AST t, together determining the mapping $v \mapsto t(\Gamma, v)$.

45b ⟨*env.h typedefs* 44b⟩+≡ (44a) ◁45a
```
typedef struct {
  env_t *       ctx;
  ast_t *       code;
} closure_t;
```

Having determined the node types, we can fill in their corresponding data fields in an environment. Note in particular the representation of pairs by a pointer to a list of children, similarly to how we structured AST's.

45c ⟨*env_s union fields* 45c⟩≡ (44c)
```
int         num;
env_t *     rchild;
closure_t   cl;
```

Every node has at least a type, so that we can make it a required argument to pass in when allocating a new instance.

45d ⟨*env.h function prototypes* 45d⟩≡ (44a) 46a▷
```
extern env_t *    Env_New(envType_t);
```

A 0-tuple is determined completely by its type, and we export a specialized macro for its creation.

45e ⟨*env.h macros* 45e⟩≡ (44a)
```
#define Env_Nil()  Env_New(ENV_NIL)
```

For convenience, we similarly export dedicated methods for constructing the other node types. `Env_New` itself, however, will still prove useful in those cases where we may not know in advance what to set the data fields with.

46a ⟨*env.h function prototypes* 45d⟩+≡ (44a) ◁45d 46b▷
```
extern env_t *    Env_Int(const int);
extern env_t *    Env_Pair(env_t * const, env_t * const);
extern env_t *    Env_Closure(env_t * const, ast_t * const);
```

Creating a copy of an environment results in a new instance that shares only its references to AST's with the original.

46b ⟨*env.h function prototypes* 45d⟩+≡ (44a) ◁46a 46c▷
```
extern env_t *    Env_Copy(const env_t * const);
```

Rather than leaving the client with the responsibility of cleaning up an environment node by node, we instead export two methods for releasing them in their entirety: one for freeing a single environment referenced by its root, in particular leaving its siblings, if any, untouched, and the other for freeing a list of environments.

46c ⟨*env.h function prototypes* 45d⟩+≡ (44a) ◁46b
```
extern void       Env_Free(env_t ** const);
extern void       Env_FreeList(env_t ** const);
```

3.2.2 Implementation

The chosen representation of environments bears much similarity to that adopted for AST's, and so many of our considerations in implementing the algorithms from the previous section will carry over for what is to follow.

46d ⟨*env.c* 46d⟩≡
```
#include "env.h"

#include <assert.h>

#include "pool.h"

⟨env.c global variables 46e⟩
⟨env.c function definitions 47a⟩
```

Like an AST, environments must be allocated from the heap, thus requiring their own memory pool.

46e ⟨*env.c global variables* 46e⟩≡ (46d)
```
static env_t    g_pool[N_ELEMS];
pool_t          g_env_pool = INIT_POOL(g_pool, N_ELEMS, env_t);
```

In creating a new node, we make sure again to clear all its bits before using it.

47a ⟨*env.c function definitions* 47a⟩≡ (46d) 47b▷
```
env_t *
Env_New(envType_t type)
{
  env_t * me = Pool_Calloc(&g_env_pool);
  me->type = type;
  return me;
}
```

In a similar vein, we can create new nodes of type `ENV_INT`.

47b ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁47a 47c▷
```
env_t *
Env_Int(const int num)
{
  env_t * me = Pool_Calloc(&g_env_pool);
  me->type = ENV_INT;
  me->u.num = num;
  return me;
}
```

In studying the creation of a pair, recall its projections are contained in a list.

47c ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁47b 48a▷
```
env_t *
Env_Pair(env_t * const left, env_t * const right)
{
  env_t *  me;

  assert(left);
  assert(right);

  me = Pool_Calloc(&g_env_pool);
  me->type = ENV_PAIR;
  Push(&me->u.rchild, right);
  Push(&me->u.rchild, left);
  return me;
}
```

Finally, we should be able to create closures.

48a ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁47c 48b▷

```
env_t *
Env_Closure(env_t * const ctx, ast_t * const code)
{
  env_t *  me;

  assert(ctx);
  assert(code);

  me = Pool_Calloc(&g_env_pool);
  me->type = ENV_CLOSURE;
  me->u.cl.ctx = ctx;
  me->u.cl.code = code;
  return me;
}
```

To copy an environment, we start with the root and switch on the latter's type to determine which of its fields to copy.

48b ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁48a 49d▷

```
env_t *
Env_Copy(const env_t * const me)
{
  env_t *  copy;

  assert(me);

  copy = Env_New(me->type);
  switch(me->type) {
  ⟨Env_Copy cases 48c⟩
  default:
    assert(0);
  }
  return copy;
}
```

If the root node was a 0-tuple, we are already done.

48c ⟨Env_Copy *cases* 48c⟩≡ (48b) 49a▷

```
case ENV_NIL:
  break;
```

If it instead carried a numeric value, we have only but to replicate it.

49a ⟨Env_Copy *cases* 48c⟩+≡ (48b) ◁48c 49b▷
```
case ENV_INT:
  copy->u.num = me->u.num;
  break;
```

For closures, it is important to recall that we don't copy the AST.

49b ⟨Env_Copy *cases* 48c⟩+≡ (48b) ◁49a 49c▷
```
case ENV_CLOSURE:
  copy->u.cl.ctx = Env_Copy(me->u.cl.ctx);
  copy->u.cl.code = me->u.cl.code;
  break;
```

Finally, for a pair, we need to copy both its projections.

49c ⟨Env_Copy *cases* 48c⟩+≡ (48b) ◁49b
```
case ENV_PAIR:
  Push(&copy->u.rchild, Env_Copy(me->u.rchild));
  Push(&copy->u.rchild, Env_Copy((env_t *)Link(me->u.rchild)));
  break;
```

To free an environment, we again first flatten it's tree structure into a list. This time, however, we shall want to accept as input not just a single environment but rather a multitude thereof, strung together into a list already.

49d ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁48b 50a▷
```
static node_t *
Flatten(env_t * me)
{
  env_t * it;

  assert(me);

  it = Link(me);
  do {
    assert(it);
    if (it->type == ENV_PAIR) {
      Append(&me, it->u.rchild);
    } else if (it->type == ENV_CLOSURE) {
      Append(&me, it->u.cl.ctx);
    }
  } while ((it = Link(it)) != Link(me));

  return (node_t *)me;
}
```

To deallocate a single environment, we first make it into a singleton list prior to flattening it.

50a ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁49d 50b▷

```
void
Env_Free(env_t ** const me)
{
  assert(me);

  if (*me == NULL) {
    return;
  }
  (*me)->base.link = (node_t *)*me;
  Pool_FreeList(&g_env_pool, Flatten(*me));
  *me = NULL;
}
```

In addition, we now also admit releasing a list of environments all at once.

50b ⟨*env.c function definitions* 47a⟩+≡ (46d) ◁50a

```
void
Env_FreeList(env_t ** const me)
{
  assert(me);

  if (*me == NULL) {
    return;
  }
  Pool_FreeList(&g_env_pool, Flatten(*me));
  *me = NULL;
}
```

3.3 The Categorical Abstract Machine

We motivated AST's using an algebraic notation for representing functions, explaining how their computation corresponded to evaluating a term inside a binding environment. The notational device we used forms part of a branch of mathematics known as category theory. While its further exposition would take us too far afoot, our interest in mentioning it derives from it being the namesake for Cousineau et al.'s Categorical Abstract Machine [1] (or CAM, for short), formalizing our intuitive explanations about evaluation using a language of machine instructions. In this section, we will implement the CAM as a tree walk over AST's, s.t. each instruction corresponds to the visitation of a node.

3.3.1 Interface

51a ⟨*cam.h* 51a⟩≡

```
#ifndef CAM_H_
#define CAM_H_

#include <stdbool.h>

#include "ast.h"
#include "env.h"

⟨cam.h typedefs 51b⟩
⟨cam.h function prototypes 52a⟩

#endif /* CAM_H_ */
```

At minimum, the CAM's state should comprise an AST and a binding environment to evaluate it in, noting the visitor methods supply the former if we implement the CAM's operation as a tree walk. It turns out this is almost enough, and that we need but one extra component. Specifically, recall the evaluation of pairings $\langle f, g \rangle$ in an environment Γ is defined in terms of the independent evaluations of its projections f and g in Γ. To implement this procedure by a sequential machine, we must choose which to evaluate first, say, f, and to store a copy of Γ somewhere temporarily until we are ready to continue with g. Since function pairings can nest, it follows we may have to provide temporary storage for more than a single environment at a time on a last-in first-out basis.

51b ⟨*cam.h typedefs* 51b⟩≡ (51a)

```
typedef struct {
  visit_t   base;
  env_t *   env;
  env_t *   stack;
} cam_t;
```

Instances of the CAM are always allocated on the stack, though requiring initialization. During its operation, however, resources may be acquired dynamically for building environments, which we must clean up again afterwards.

52a ⟨*cam.h function prototypes* 52a⟩≡ (51a)
```
extern void Cam_Init(cam_t * const);
extern void Cam_Free(cam_t * const);
```

3.3.2 Implementation

52b ⟨*cam.c* 52b⟩≡
```
#include "cam.h"

#include <assert.h>

#include "pool.h"

⟨cam.c function prototypes 52c⟩
⟨cam.c function definitions 52d⟩
```

The CAM's instruction set is implemented by visitor methods, each of which we will explain in turn below.

52c ⟨*cam.c function prototypes* 52c⟩≡ (52b)
```
static statusCode_t VisitFst(cam_t * const, const ast_t *);
static statusCode_t VisitSnd(cam_t * const, const ast_t *);
static statusCode_t VisitQuote(cam_t * const, const ast_t *);
static statusCode_t VisitApp(cam_t * const, const ast_t *);
static statusCode_t VisitCur(cam_t * const, const ast_t *);
static statusCode_t VisitPush(cam_t * const, const ast_t *);
static statusCode_t VisitSwap(cam_t * const, const ast_t *);
static statusCode_t VisitCons(cam_t * const, const ast_t *);
static statusCode_t VisitPlus(cam_t * const, const ast_t *);
```

Initialisation sets the virtual function table, the environment and the stack.

52d ⟨*cam.c function definitions* 52d⟩≡ (52b) 53c▷
```
void Cam_Init(cam_t * const me)
{
  ⟨define cam_t virtual function table 53a⟩

  assert(me);

  ⟨initialize cam_t state 53b⟩
}
```

The reasons for separating out a virtual function table from the definition of a visitor was to allow for sharing of the former's instances. As such, we define the table used for our implementation of the CAM by a static variable, guaranteeing the same object is accessed among multiple invocations of `Cam_Init`. Note further we need explicit casts from our custom visitor methods to `visitFunc_t` because of their first argument being typed `cam_t`. Since, however, every `cam_t` 'is a' `visitor_t` (in the reading commonly attributed to this phrase in the Object-Oriented paradigm), this is safe.

53a ⟨*define* `cam_t` *virtual function table* 53a⟩≡ (52d)

```
static const visitVtbl_t vtbl = {
                VisitDefault,   /* VisitId */
  (visitFunc_t) VisitApp,       /* VisitApp */
  (visitFunc_t) VisitQuote,     /* VisitQuote */
  (visitFunc_t) VisitPlus,      /* VisitPlus */
  (visitFunc_t) VisitFst,       /* VisitFst */
  (visitFunc_t) VisitSnd,       /* VisitSnd */
                VisitDefault,   /* PreVisitComp */
  (visitFunc_t) VisitPush,      /* PreVisitPair */
  (visitFunc_t) VisitCur,       /* PreVisitCur */
  (visitFunc_t) VisitSwap,      /* InVisitPair */
                VisitDefault,   /* PostVisitComp */
  (visitFunc_t) VisitCons,      /* PostVisitPair */
                VisitDefault    /* PostVisitCur */
};
```

As for the CAM's state, we start out with a clean slate by using a 0-tuple for its environment together with an empty stack.

53b ⟨*initialize* `cam_t` *state* 53b⟩≡ (52d)

```
me->env = Env_Nil();
me->stack = NULL;
me->base.vptr = &vtbl;
```

Before retiring one of the CAM's instances, we first have to clean up its environment and stack, both having been dynamically allocated.

53c ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁52d 54a▷

```
void Cam_Free(cam_t * const me)
{
  Env_Free(&me->env);
  Env_FreeList(&me->stack);
}
```

We ease into our exposition of the CAM's instruction set with the interpretation of constants. Recall that given a non-negative integer c, we have $('c)(\Gamma) = c$ for any environment Γ. We can translate this to an instruction QUOTE, applied to an AST of type `AST_INT`, and having the effect of discarding the current environment and replacing it with a single numeric constant. Note in particular how we stored the results of our evaluation. In general, we shall write our code to observe the following invariants. After a node f's postvisiting, the CAM's environment will consist of the single value obtained from computing $f(\Gamma)$, where Γ was its environment prior to f's previsiting. In addition, after f's traversal, the stack will be (back) in the same state as before. Both these invariants should be kept firmly in mind when reading the explanations and code to come.

54a ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁53c 54b▷

```
static statusCode_t
VisitQuote(cam_t * const me, const ast_t *ap)
{
  Env_Free(&me->env);
  me->env = Env_Int(ap->value);

  return SC_CONTINUE;
}
```

Given $\langle f, g \rangle$, we can read each of '$\langle$', ',' and '$\rangle$' as separate machine instructions, coinciding, respectively, with pre-, in- and postvisiting an AST of type `AST_PAIR`. The first pushes a copy of the environment Γ on the stack, earning it the name PUSH.

54b ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁54a 55a▷

```
static statusCode_t
VisitPush(cam_t * const me, const ast_t *ap)
{
  (void)ap;

  Push(&me->stack, Env_Copy(me->env));

  return SC_CONTINUE;
}
```

Prior to 'in'visiting $\langle f, g\rangle$, the machine is in a state where its environment contains the result of computing $f(\Gamma)$ for some Γ, and with a copy of the latter at the head of its stack. Our next task shall be to compute $g(\Gamma)$, thus requiring to pop Γ off the stack again to set the machine's environment therewith. At the same time we still have to retain our previous result, needing it again for when we're done traversing g. As it turns out, the stack provides the perfect spot for safekeeping, reducing our task to that of simply SWAPping the CAM's environment with the top of its stack.

55a ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁54b 55b▷

```
static statusCode_t
VisitSwap(cam_t * const me, const ast_t *ap)
{
  env_t *  tmp;

  (void)ap;
  assert(me->stack != NULL);

  tmp = Pop(&me->stack);
  Push(&me->stack, me->env);
  me->env = tmp;

  return SC_CONTINUE;
}
```

As we are about to postvisit $\langle f, g\rangle$, we have $g(\Gamma)$ set as our environment and $f(\Gamma)$ at the top of the stack. To finish the job, we must pop the stack and replace the environment with $(f(\Gamma), g(\Gamma))$, referring to the corresponding instruction by CONS.

55b ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁55a 56a▷

```
static statusCode_t
VisitCons(cam_t * const me, const ast_t *ap)
{
  (void)ap;
  assert(me->stack != NULL);

  me->env = Env_Pair((env_t *)Pop(&me->stack), me->env);

  return SC_CONTINUE;
}
```

Remember *Fst* always takes as argument a *pair* (x, y), returning x. Similarly, in its visiting we shall assume the CAM's environment to be a pair as well, stating this as a precondition. While extracting the first projection should hardly pose a challenge, we do have to be careful in ensuring both its sibling and its parent are properely cleaned up again.

56a ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁55b 56b▷

```
static statusCode_t
VisitFst(cam_t * const me, const ast_t *ap)
{
  env_t *  proj;

  (void)ap;
  assert(me->env->type == ENV_PAIR);

  proj = Pop(&me->env->u.rchild);
  proj->base.link = NULL; /* prevent dangling pointer */
  Env_Free(&me->env);
  me->env = proj;

  return SC_CONTINUE;
}
```

The same considerations discussed for *Fst* apply to *Snd* as well, resulting in largely similar code.

56b ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁56a 57a▷

```
static statusCode_t
VisitSnd(cam_t * const me, const ast_t *ap)
{
  env_t *  proj;

  (void)ap;
  assert(me->env->type == ENV_PAIR);

  proj = me->env->u.rchild;
  Env_Free((env_t **)&proj->base.link);
  Pool_Free(&g_env_pool, (node_t *)me->env);
  me->env = proj;

  return SC_CONTINUE;
}
```

In visiting an abstraction $\Lambda(f)$ with an environment Γ, we replace the latter with a closure, determining a mapping $v \mapsto f(\Gamma, v)$. It follows that we cannot yet walk f until its second argument v is known, meaning we have to skip it for now. As explained before, the operator Λ essentially amounts to Currying, motivating the name CUR for referring to the current instruction.

57a ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁56b 57b▷

```
static statusCode_t
VisitCur(cam_t * const me, const ast_t *ap)
{
  me->env = Env_Closure(me->env, ap->rchild);

  return SC_SKIP;
}
```

In visiting *App*, we state the precondition(s) that the environment is a pair whose first projection is a closure formed from f and Γ. We proceed by computing $f(\Gamma, v)$ for v the second projection, meaning we traverse f with the environment set to (Γ, v).

57b ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁57a 58▷

```
static statusCode_t
VisitApp(cam_t * const me, const ast_t *ap)
{
  env_t *   closure;

  (void)ap;
  assert(me->env->type == ENV_PAIR);

  closure = Pop(&me->env->u.rchild);
  assert(closure->type == ENV_CLOSURE);

  Push(&me->env->u.rchild, closure->u.cl.ctx);
  Ast_Traverse(closure->u.cl.code, (visit_t *)me);
  Pool_Free(&g_env_pool, (node_t *)closure);

  return SC_CONTINUE;
}
```

In visiting +, we assume the environment to be set to (m, n) for non-negative integers m, n, replacing it with $m+n$. The details, however, get a bit messy in the reuse of (environment) nodes to minimize cleanup and prevent new allocations.

58 ⟨*cam.c function definitions* 52d⟩+≡ (52b) ◁57b

```
static statusCode_t
VisitPlus(cam_t * const me, const ast_t *ap)
{
  env_t *  left;
  env_t *  right;

  (void)ap;

  assert(me->env->type == ENV_PAIR);
  left = Pop(&me->env->u.rchild);
  assert(left->type == ENV_INT);
  right = Pop(&me->env->u.rchild);
  assert(right->type == ENV_INT);

  left->u.num += right->u.num;
  left->base.link = NULL; /* prevent dangling pointer */
  Pool_Free(&g_env_pool, (node_t *)right);
  Pool_Free(&g_env_pool, (node_t *)me->env);
  me->env = left;

  return SC_CONTINUE;
}
```

3.4 Optimization

Our discussion so far emphasized λ-terms as a notational device for unifying various algebraic modes of expression adopted across different branches of mathematics and computer science, furthermore explaining how they could be evaluated to the objects of a scientist's domain of discourse, like numbers. Quite another perspective exists as to their utility, however, viewing them as a means of performing arbitrary calculations. Backed by a system of rewriting rules, they enable a model of computation equivalent to that embodied by Turing Machines; a result that would form the foundation for the Church-Turing thesis.

Our goals for this section are not to develop the full theory of term rewriting in its applications to the λ-calculus, but rather to motivate it briefly and show how its results can be used for the purpose of developing optimization passes over AST's prior to their evaluation, developing an idea in Cousinea et al. [1]. To start with a concrete example, consider the following definitional equation:

$$x + 2 \textbf{ where } x = 1$$

In λ-calculus, we may alternatively render this expression by the term $((\lambda x.(+\ x\ 2))\ 1)$. It should be clear, intuitively, that the latter does not differ in its meaning from the result of substituting 1 for x in $(+\ x\ 2)$, i.e., $(+\ 1\ 2)$. To generalize this result, as well as making it more explicit, consider the AST $App \circ \langle \Lambda(f), g \rangle$ for the application of an abstraction to some argument. Applied to a context Γ, we obtain the following sequence of computations

$$\begin{aligned}
& (App \circ \langle \Lambda(f), g \rangle)(\Gamma) \\
= \ & App(\langle \Lambda(f), g \rangle(\Gamma)) \\
= \ & App(\langle \Lambda(f)(\Gamma), g(\Gamma) \rangle) \\
= \ & \Lambda(f)(\Gamma)(g(\Gamma)) \\
= \ & f(\Gamma, g(\Gamma)).
\end{aligned}$$

motivating the 'equivalence' of our original term $App \circ \langle \Lambda(f), g \rangle$ and $f \circ \langle Id, g \rangle$. By a similar line of reasoning, we can infer $Fst \circ \langle f, g \rangle$ may always be replaced by f, and $Snd \circ \langle f, g \rangle$ by g. More such equivalences exist, and their systematic study is a cornerstone of category theory. When related specifically to λ-terms, they bear similarity to calculi of explicit substitutions, as has been explored, among others, by Curien [3], and, though never stated there explicitly, also always being felt throughout De Bruijn's [5] exposition of his nameless notation. As for the current work, we shall not dive any deeper into such matters than we already have, and will rather limit ourselves to only the three equivalences motivated above, showing how we can implement optimization passes therewith.

3.4.1 Interface

We implement our optimizer as a treewalk over an AST, leaving the latter unmodified while producing an entirely new copy.

60a ⟨*optim.h* 60a⟩≡

```
#ifndef OPTIM_H_
#define OPTIM_H_

#include "ast.h"

⟨optim.h typedefs 60b⟩
⟨optim.h function prototypes 60c⟩

#endif /* OPTIM_H_ */
```

During an optimization pass, we keep track in a variable `cnt` of the number of transformations that were applied, allowing us later to apply multiple passes in iteration until no more changes were made. In addition, we will maintain a stack of freshly allocated nodes satisfying the invariant that during a postvisit of some f, repeatedly popping the stack will produce the AST's built during the traversals of f's children in right-to-left order, followed by a copy made of their parent node during the latter's previsit. These components we can then combine into an AST representing the result of optimizing f as a whole, which we then push back on the stack.

60b ⟨*optim.h typedefs* 60b⟩≡ (60a)

```
typedef struct {
  visit_t   base;
  node_t *  stack;
  int       cnt;
} optim_t;
```

Like instances of our evaluator, those of our optimizers are allocated only on the stack, requiring separate initialization. While heap-allocated resources áre acquired during the operation of an optimizer, all these are accessible afterwards as part of the AST remaining (alone) on the optimizer's stack. As such, we do not require an additional cleanup method, seeing as we would rather keep our results for further processing as opposed to immediately disposing of them again.

60c ⟨*optim.h function prototypes* 60c⟩≡ (60a)

```
extern void Optim_Init(optim_t * const);
```

3.4.2 Implementation

61a ⟨*optim.c* 61a⟩≡

```
#include "optim.h"

#include <assert.h>
#include <stdbool.h>
#include <stddef.h>

#include "pool.h"

⟨optim.c function prototypes 61b⟩
⟨optim.c function definitions 61c⟩
```

Being implemented as a walker, the algorithm applied by the optimizer is distributed over its visitor methods. Notice, again, the mismatch in method signatures with respect to the type declared for the first argument when compared to the definition of `visitFunc_t`. Given that every `optim_t` 'is a' `visitor_t`, however, we can safely resolve the matter later on using explicit casts.

61b ⟨*optim.c function prototypes* 61b⟩≡ (61a)

```
static statusCode_t PreVisitParent(optim_t * const, const ast_t *);
static statusCode_t PostVisitParent(optim_t * const, const ast_t *);
static statusCode_t VisitLeaf(optim_t * const, const ast_t *);
static statusCode_t VisitFst(optim_t * const, const ast_t *);
static statusCode_t VisitSnd(optim_t * const, const ast_t *);
static statusCode_t VisitApp(optim_t * const, const ast_t *);
```

Again similar to our prior exposition of the CAM, we initialize an optimizer pass by setting the virtual function table as well as its count and stack.

61c ⟨*optim.c function definitions* 61c⟩≡ (61a) 62b▷

```
void
Optim_Init(optim_t * const me)
{
  ⟨define optimizer virtual function table vtbl 62a⟩

  assert(me);

  me->stack = NULL;
  me->cnt = 0;
  me->base.vptr = &vtbl;
}
```

The virtual function table is again shared between different optimizer instances.

62a ⟨*define optimizer virtual function table* `vtbl` 62a⟩≡ (61c)

```
static const visitVtbl_t vtbl = {
  (visitFunc_t) VisitLeaf,        /* VisitId */
  (visitFunc_t) VisitApp,         /* VisitApp */
  (visitFunc_t) VisitLeaf,        /* VisitQuote */
  (visitFunc_t) VisitLeaf,        /* VisitPlus */
  (visitFunc_t) VisitFst,         /* VisitFst */
  (visitFunc_t) VisitSnd,         /* VisitSnd */
  (visitFunc_t) PreVisitParent,   /* PreVisitComp */
  (visitFunc_t) PreVisitParent,   /* PreVisitPair */
  (visitFunc_t) PreVisitParent,   /* PreVisitCur */
                VisitDefault,     /* InVisitPair */
  (visitFunc_t) PostVisitParent,  /* PostVisitComp */
  (visitFunc_t) PostVisitParent,  /* PostVisitPair */
  (visitFunc_t) PostVisitParent   /* PostVisitCur */
};
```

When popping nodes off the optimizer's stack, how do we tell siblings from their parent? We must clearly mark the latter, and we do so by making it its own parent. Indeed, while created during a node's previsit, we won't set its children until the postvisit, guaranteeing `rchild` isn't being used anyway until then.

62b ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁61c 62c▷

```
static inline bool
IsSibling(const ast_t * const ap)
{
  return ap->rchild != ap;
}
```

When previsiting a parent node, we push a copy thereof on the stack, making sure to mark it.

62c ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁62b 63a▷

```
static statusCode_t
PreVisitParent(optim_t * const me, const ast_t *ap)
{
  ast_t * copy;

  (void)ap;
  copy = Ast_Node(ap->type);
  copy->rchild = copy;
  Push(&me->stack, copy);
  return SC_CONTINUE;
}
```

In postvisiting a node, we first pop the AST's off the stack built during the traversals of its children, followed by a copy of the parent node. After combining them into a single tree, we push the result back on the stack.

63a ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁62c 64c▷

```
static statusCode_t
PostVisitParent(optim_t * const me, const ast_t *ap)
{
  ast_t *   head;
  ast_t *   children = NULL;

  ⟨pop children and set head to parent 63b⟩

  Ast_SetChildren(head, children);

  ⟨replace empty composition with identity 64b⟩

  Push(&me->stack, head);

  return SC_CONTINUE;
}
```

Popping the child nodes produces them in the wrong (i.e., right-to-left) order. To counteract, we immediately push them onto a separate (initially empty) stack, which we afterwards use in setting the child list of their parent, popped immediately afterwards.

63b ⟨*pop* children *and set* head *to parent* 63b⟩≡ (63a)

```
for (;;) {
  head = Pop(&me->stack);
  assert(head);
  if (!IsSibling(head)) {
    /* head must be a parent */
    break;
  }
  if (ap->type == AST_COMP) {
    ⟨observe associativity and identity laws for composition 64a⟩
  }
  Push(&children, head);
}
```

If the postvisited node is a composition, then based on its associativity we take special care to avoid adding a child node that is a composition itself, rather preferring to add the latter's children directly. Note we can safely assume that any such child compositions have themselves already been flattened during a previous postvisit, easing our task. In addition, we can entirely omit adding child nodes of type `AST_ID` by virtue of the identity law.

64a ⟨*observe associativity and identity laws for composition* 64a⟩≡ (63b)

```
switch (head->type) {
case AST_COMP:
  Prepend(&children, head->rchild);
  /* Fall-through */
case AST_ID:
  Pool_Free(&g_ast_pool, (node_t *)head);
  ++me->cnt;
  continue;
default:
  break;
}
```

By not adding child nodes of type `AST_ID` when constructing a composition, the latter may end up with no children at all. In this case, we replace it entirely with a node of type `AST_ID`.

64b ⟨*replace empty composition with identity* 64b⟩≡ (63a)

```
if (head->type == AST_COMP && head->rchild == NULL) {
  head->type = AST_ID;
}
```

In most cases, we process a leaf node by simply copying it. In the paragraphs to come, we will discuss the exceptions to this rule one by one.

64c ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁63a 65a▷

```
static statusCode_t
VisitLeaf(optim_t * const me, const ast_t *ap)
{
  ast_t * copy;

  copy = Ast_Node(ap->type);
  copy->value = ap->value;
  Push(&me->stack, copy);

  return SC_CONTINUE;
}
```

Upon visiting *Fst*, we verify if its sibling is a pair. If not, we resort to the default behaviour of simply copying it. Otherwise, we replace said pair with its first projection.

65a ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁64c 65c▷
```
static statusCode_t
VisitFst(optim_t * const me, const ast_t *ap)
{
  ast_t * head;

  if ((head = Peek(me->stack)) && IsSibling(head)
        && head->type == AST_PAIR) {
    ⟨replace Fst ∘ ⟨f, g⟩ with f 65b⟩
  }
  return VisitLeaf(me, ap);
}
```

Our effective implementation of term transformations is rather naive, in that we often end up discarding part of our previous work. In replacing a pair with its first projection, for instance, we already built the former in its entirety only to now deallocate both the parent node and its right projection again.

65b ⟨*replace Fst* ∘ ⟨*f*, *g*⟩ *with f* 65b⟩≡ (65a)
```
Pop(&me->stack);
Push(&me->stack, Pop(&head->rchild));
Ast_Free(&head);
++me->cnt;
return SC_CONTINUE;
```

The actions applied at visiting *Snd* should hold little surprise after having already studied the case of *Fst*.

65c ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁65a 66b▷
```
static statusCode_t
VisitSnd(optim_t * const me, const ast_t *ap)
{
  ast_t * head;

  if ((head = Peek(me->stack)) && IsSibling(head)
        && head->type == AST_PAIR) {
    ⟨replace Snd ∘ ⟨f, g⟩ with g 66a⟩
  }
  return VisitLeaf(me, ap);
}
```

The cleanup performed in the replacement of a pair with its second projection requires slightly more work compared to before, but otherwise follows much the same pattern.

66a ⟨*replace* $Snd \circ \langle f, g \rangle$ *with* g 66a⟩≡ (65c)

```
Pop(&me->stack);
Ast_Free((ast_t **)&head->rchild->base.link);
Push(&me->stack, head->rchild);
Pool_Free(&g_ast_pool, (node_t *)head);
++me->cnt;
return SC_CONTINUE;
```

We conclude with our last transformation, concerning the replacement of $App \circ \langle \Lambda(f), g \rangle$ with $f \circ \langle Id, g \rangle$. Again, said transformation is triggered upon visiting *App* when its sibling is a pair, otherwise defaulting to the mere creation of a copy.

66b ⟨*optim.c function definitions* 61c⟩+≡ (61a) ◁65c

```
static statusCode_t
VisitApp(optim_t * const me, const ast_t *ap)
{
  ast_t * head;
  ast_t * left;

  if ((head = Peek(me->stack)) && IsSibling(head)
      && head->type == AST_PAIR && (left = Peek(head->rchild))
      && left->type == AST_CUR) {
    ⟨replace App ∘ ⟨Λ(f), g⟩ with f ∘ ⟨Id, g⟩ 66c⟩
  }
  return VisitLeaf(me, ap);
}
```

As before, the gory details reveal a naive approach, favouring redundant work.

66c ⟨*replace* $App \circ \langle \Lambda(f), g \rangle$ *with* $f \circ \langle Id, g \rangle$ 66c⟩≡ (66b)

```
Pop(&head->rchild);
Ast_AddChild(head, Ast_Id());
Push(&me->stack, left->rchild);
Pool_Free(&g_ast_pool, (node_t *)left);
++me->cnt;
return SC_CONTINUE;
```

Chapter 4

The Read-Eval Print Loop

Having implemented both the interpreter and optimizer as tree traversals over AST's, we now proceed to the writing of a user interface for exercising them. Specifically, we shall provide a Read-Eval-Print Loop (REPL) accepting closed λ-terms denoting a number, which are then tokenized and parsed into an AST before being evaluated.

In §4.1, we first provide a grammar for the specific fragment of λ-calculus that we shall support, adopting a syntax similar to that of LISP (though that's where the similarities end). We continue with tokenizing in §4.2 and parsing in §4.3, concluding with the REPL itself in §4.4.

4.1 Concrete syntax

Figure 4.1 defines a grammar in Extended Backus-Naur Form (EBNF) for the input format that we shall accept. Though we assume familiarity on the reader's part with context-free grammars, our notation, conforming to the ISO standard [8], may require a few words of explanation. The distinction between terminal- and non-terminal symbols is one of the absence, resp. presence of surrounding double quotes. Comma denotes concatenation, and alternatives are indicated by vertical bars. Rules are terminated by a semicolon, and, finally, repetition (in the sense of 0 or more) is denoted using curly brackets.

The language we defined does not admit the full generality that ordinary λ-calculus provides, and moreover defines but one operator constant. Little stands in the reader's way of extending the current codebase to rememdy the latter limitation, although the first is more serious. Specifically, to make our life easier, we have restricted to the description of expressions all whose constituents we are certain denote numbers. This obviates the need for type checking, but makes it impossible to abstract over functions. To illustrate, let us consider some concrete expressions that may be formed using the above grammar. First,

```
((lambda (x y) (+ 1 x y)) 2 (+ 3 4))
```

```
expr  = var | num | sum | app ;
num   = digit, { digit } ;
var   = alpha, { alpha } ;
sum   = "(", "+", expr, { expr }, ")" ;
app   = "(", abs, expr, { expr }, ")" ;
abs   = "(", "lambda", "(", var, { var }, ")", expr, ")" ;
alpha = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
      | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
      | "U" | "W" | "X" | "Y" | "Z"
      | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
      | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
      | "u" | "w" | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Figure 4.1: Input format in EBNF.

Landin [12] explains expressions like the above as an encoding of auxiliary definitions, e.g.,

$$1 + x + y \textbf{ where } x = 2 \textbf{ and } y = 3 + 4$$

Notice in particular some of the shorthands we have admitted ourselves. Whereas previously we only allowed abstraction over a single variable at a time, above we seemingly bound both x and y to the same λ. The way we shall resolve this seeming discrepancy in what is to follow is by considering the current example an abbreviation for two separate bindings, one of x and the other of y. Similarly, if we assume application associates to the left, we can simply write, e.g., $(+\ 1\ x\ y)$ instead of $((+\ 1\ x)\ y)$.

Besides cases like the above, Landin also discusses how λ-calculus may be used to represent function definitions, giving, among others, the following (here simplified) example on p.311:

$$f(3) + f(4) \textbf{ where } f(y) = y + 1$$

An attempt at expressing this in our input language may run as follows:

```
((lambda (f) (+ (f 3) (f 4))) (lambda (y) (+ y 1)))
```

were it not for the fact that our grammar only permits the appearance of an abstraction in the first sub-expression of an application, which the above violates. As mentioned, this limitation was part of a conscious design decision to keep the parser as simple as possible, allowing to focus more on the CAM's implementation while enabling the project's completion within a reasonable timeframe.

4.2 The lexer

4.2.1 Interface

The lexer's task is to tokenize the input stream, relieving the parser from the burden of dealing with matters of formatting by grouping non-whitespace characters into meaningful chunks.

69a ⟨*lexer.h* 69a⟩≡

```
#ifndef LEXER_H_
#define LEXER_H_

⟨lexer.h constants 70a⟩
⟨lexer.h typedefs 69b⟩
⟨lexer.h function prototypes 70b⟩

#endif /* LEXER_H_ */
```

Token types are encoded by integers, those categorizing single characters being assigned the latter's ASCII value. In addition, we created a 'sentinel' value to be used in case of errors or when the end of the input has been reached.

69b ⟨*lexer.h typedefs* 69b⟩≡ (69a) 69c▷

```
typedef enum {
  /* multi-character tokens */
  LEX_LAMBDA = 1,   /* "lambda" */
  LEX_VAR    = 2,   /* e.g., "foo", "Bar" */
  LEX_NUM    = 3,   /* integers */

  /* single-character tokens */
  LEX_LBRACK = 40,  /* '(' */
  LEX_RBRACK = 41,  /* ')' */
  LEX_PLUS   = 43,  /* '+' */

  /* special value for errors and end-of-string */
  LEX_NONE   = 0,
} tokenType_t;
```

A lexer's state includes a buffer for storing the recognized token, together with its type. An additional pointer stores the position in the input string.

69c ⟨*lexer.h typedefs* 69b⟩+≡ (69a) ◁69b

```
typedef struct lexer_s {
  char          token[MAXTOK + 1];
  tokenType_t   type;
  const char *  ptr;
} lexer_t;
```

We imposed a maximum token size of 10, having reserved an additional character in the input buffer for storing the terminating `'\0'`.

70a ⟨*lexer.h constants* 70a⟩≡ (69a)
```
enum {
  MAXTOK = 10
};
```

Lexer instances are allocated on the stack and passed to an initialization method via a pointer. The latter additionally takes a reference to the beginning of the input, which will reside in an in-memory string buffer filled by the REPL.

70b ⟨*lexer.h function prototypes* 70b⟩≡ (69a) 70c▷
```
extern void   Lexer_Init(lexer_t * const, const char * const);
```

Retrieving the next token sets the lexer's buffer and token type, returning the number of (non-whitespace) characters read. The end of the input is indicated by `0`, while `-1` signals an error.

70c ⟨*lexer.h function prototypes* 70b⟩+≡ (69a) ◁70b
```
extern int    Lexer_NextToken(lexer_t * const);
```

4.2.2 Implementation

70d ⟨*lexer.c* 70d⟩≡
```
#include "lexer.h"

#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

⟨lexer.c function definitions 70e⟩
```

The lexer is initialized by clearing its buffer and setting the start of the input.

70e ⟨*lexer.c function definitions* 70e⟩≡ (70d) 71a▷
```
void
Lexer_Init(lexer_t * const me, const char * const input)
{
  assert(me);
  assert(input);

  me->token[0] = '\0';
  me->type = LEX_NONE;
  me->ptr = input;
}
```

To get a token we switch on the next non-whitespace character in the input. At this point, we also start copying any character we read to the lexer's buffer.

71a ⟨*lexer.c function definitions* 70e⟩+≡ (70d) ◁70e

```
int
Lexer_NextToken(lexer_t * const me)
{
  char *  cp;

  assert(me);

  for (; isspace(*me->ptr); ++me->ptr)
    ;
  cp = me->token;
  switch (*me->ptr) {
  ⟨NextToken cases 71b⟩
  default:
    ⟨NextToken error handling 72b⟩
  }
  ⟨NextToken return character count 72c⟩
}
```

The input is terminated by the same character `'\0'` delimiting strings in C.

71b ⟨*NextToken cases* 71b⟩≡ (71a) 71c▷

```
case '\0':
  me->type = LEX_NONE;
  break;
```

Recall single-character tokens coincide with the value of their token type.

71c ⟨*NextToken cases* 71b⟩+≡ (71a) ◁71b 71d▷

```
case '+': case '(': case ')':
  me->type = *cp++ = *me->ptr++;
  break;
```

Integers are recognized by continuing to read digits until the token buffer is full.

71d ⟨*NextToken cases* 71b⟩+≡ (71a) ◁71c 72a▷

```
case '0': case '1': case '2': case '3': case '4': case '5':
case '6': case '7': case '8': case '9':
  do {
    *cp++ = *me->ptr++;
  } while (cp - me->token < MAXTOK && isdigit(*me->ptr));
  me->type = LEX_NUM;
  break;
```

The logic for recognizing variable names bears strong resemblance to that of integers, although we must remember to check for the keyword `lambda`.

72a ⟨*NextToken cases* 71b⟩+≡ (71a) ◁71d

```
case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
case 'g': case 'h': case 'i': case 'j': case 'k': case 'l':
case 'm': case 'n': case 'o': case 'p': case 'q': case 'r':
case 's': case 't': case 'u': case 'v': case 'w': case 'x':
case 'y': case 'z':
case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
case 'G': case 'H': case 'I': case 'J': case 'K': case 'L':
case 'M': case 'N': case 'O': case 'P': case 'Q': case 'R':
case 'S': case 'T': case 'U': case 'V': case 'W': case 'X':
case 'Y': case 'Z':
  do {
    *cp++ = *me->ptr++;
  } while (cp - me->token < MAXTOK && isalpha(*me->ptr));
  me->type = (strcmp(me->token, "lambda") == 0)
    ? LEX_LAMBDA
    : LEX_VAR;
  break;
```

If the next input character cannot be the start of a valid token, we print an error message, empty the token buffer and signal an error.

72b ⟨*NextToken error handling* 72b⟩≡ (71a)

```
fprintf(stderr, "Unexpected character: %c.\n", *me->ptr);
*cp = '\0';
me->type = LEX_NONE;
++me->ptr;
return -1;
```

Once a valid token has been read in, we return its size after properly terminating the string in the token buffer.

72c ⟨*NextToken return character count* 72c⟩≡ (71a)

```
*cp = '\0';
return cp - me->token;
```

4.3 The parser

4.3.1 Interface

With the lexer converting an input string of characters into a stream of tokens, the parser next attempts the recognition of structure, recording its observations in an AST. By keeping the parser stateless, its interface remains simple as well.

73a ⟨*parser.h* 73a⟩≡

```
#ifndef PARSER_H_

#include "ast.h"
#include "lexer.h"

extern ast_t *  Parse(lexer_t * const);

#endif /* PARSER_H_ */
```

4.3.2 Implementation

We implement the parser using recursive-descent, meaning the call stack is used to trace a branch in the parse tree. Essentially, each grammar rule in Figure 4.1 translates to a method, making this an easy technique to use for writing a parser by hand with.

73b ⟨*parser.c* 73b⟩≡

```
#include "parser.h"

#include <assert.h>
#include <ctype.h>
#include <setjmp.h>
#include <stdio.h>
#include <string.h>

#include "ast.h"
#include "except.h"
#include "lexer.h"
#include "node.h"
#include "pool.h"

⟨parser.c typedefs 74a⟩
⟨parser.c global variables 74b⟩
⟨parser.c function prototypes 75a⟩
⟨parser.c function definitions 74c⟩
```

As we process a term we shall want to keep track of variable bindings. Recall that in De Bruijn's notation, names are replaced with numbers indicating their distance to the binding site. Thus, in translating a variable occurrence, we must navigate the parse tree back up to the root node, counting the λ's seen on the way until a match is found (signalling an error otherwise). We can implement this process by maintaining a stack of variable names, corresponding to the bindings in the order that we encountered them. We speak interchangeably of a *scope*, calling its individual nodes *symbols*.

74a ⟨*parser.c typedefs* 74a⟩≡ (73b)
```
typedef struct {
  node_t  base;
  char    value[MAXTOK + 1];
} symbol_t;
```

Though the lifetimes of symbols are tied to the method invocations that record a branch of the parse tree in the call stack, the fact that we allowed multiple variables to be bound at once in our input language makes it impossible to know at compile time just how many symbols to allocate upon the processing of any given λ. As such, we store symbols in a memory pool.

74b ⟨*parser.c global variables* 74b⟩≡ (73b)
```
static symbol_t g_pool[N_ELEMS];
pool_t          g_symbol_pool = INIT_POOL(g_pool, N_ELEMS, symbol_t);
```

The process of allocating and initializing a new symbol and pushing it onto a scope will be repeated sufficiently often in what is to follow as to justify its encapsulation into a separate method.

74c ⟨*parser.c function definitions* 74c⟩≡ (73b) 75b▷
```
static void
PushNewSymbol(const symbol_t ** scope, const char * const token)
{
  symbol_t *  symbol;

  assert(scope);
  assert(token);

  symbol = Pool_Alloc(&g_symbol_pool);
  strcpy(symbol->value, token);
  Push(scope, symbol);
}
```

With the exception of `alpha`, each non-terminal from Figure 4.1 has its own method. All return an AST, and most take an additional argument for the scope to resolve free variable occurences.

75a ⟨*parser.c function prototypes* 75a⟩≡ (73b)

```
static ast_t * ParseExpr(lexer_t * const, const symbol_t *);
static ast_t * ParseVar(const char * const, const symbol_t * const);
static ast_t * ParseNum(const char *);
static ast_t * ParseSum(lexer_t * const, const symbol_t *);
static ast_t * ParseApp(lexer_t * const, const symbol_t *);
static ast_t * ParseAbs(lexer_t * const, const symbol_t *, int *);
```

We use a number of helper methods for implementing the grammar rules. The first, `Consume`, simply attempts to read the next token. If none is available, we print a message. In this case, as well as when the lexer reports an error, we give up parsing immediately and throw an exception.

75b ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁74c 75c▷

```
static void
Consume(lexer_t * const lexer)
{
  int cnt;

  switch (cnt = Lexer_NextToken(lexer)) {
  case 0:
    fprintf(stderr, "Unexpected end of input.\n");
    /* fall-through */
  case -1:
    THROW;
  default:
    return;
  }
}
```

Next, `Match` allows to validate the type of the last consumed token, reporting an error and raising an exception in case of a mismatch.

75c ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁75b 76a▷

```
static void
Match(lexer_t * const lexer, const tokenType_t type)
{
  if (type != lexer->type) {
    fprintf(stderr, "Unexpected token: %s.\n", lexer->token);
    THROW;
  }
}
```

As the last of our helper methods, we combine `Consume` and `Match` into a single function `Expect`.

76a ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁75c 76b▷
```
static inline void
Expect(lexer_t * const lexer, const tokenType_t type)
{
  Consume(lexer);
  Match(lexer, type);
}
```

To start parsing, we consume the first token and invoke the start symbol with an empty scope.

76b ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁76a 76c▷
```
ast_t *
Parse(lexer_t * const lexer)
{
  Consume(lexer);
  return ParseExpr(lexer, NULL);
}
```

We start our implementation of the grammar rules with the start symbol. Recall an expression is a variable, a number, or an application.

76c ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁76b 77b▷
```
static ast_t *
ParseExpr(lexer_t * const lexer, const symbol_t *scope)
{
  assert(lexer->type != LEX_NONE);

  switch (lexer->type) {
  case LEX_VAR:
    return ParseVar(lexer->token, scope);
  case LEX_NUM:
    return ParseNum(lexer->token);
  case LEX_LBRACK:
    ⟨parse application 77a⟩
  default:
    fprintf(stderr, "Unexpected token: %s.\n", lexer->token);
    THROW;
  }
}
```

Whereas applications may be handled by a single production in the full λ-calculus, instead, in order to accommodate the restrictions discussed in §4.1, we have here had to split it up based on whether the operand coincides with `+` (cf. the rule for `sum`) or an abstraction (`app`). To differentiate between the two cases, we will need to look ahead one extra token.

77a ⟨*parse application* 77a⟩≡ (76c)

```
Consume(lexer);
if (lexer->type == LEX_PLUS) {
  return ParseSum(lexer, scope);
}
return ParseApp(lexer, scope);
```

We already briefly explained the translation of variables. Given a scope, we count the symbols as we retrace our steps to the first that we saw. If a match is found, we convert the running count n into a composition of projections, picking out the n^{th} term from the right in an environment. Else, if the variable is unbound, we print an error message and raise an exception.

77b ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁76c 78a▷

```
static ast_t *
ParseVar(const char * const token, const symbol_t * const scope)
{
  ast_t *     ap;
  symbol_t *  it;

  assert(token);

  if (IsEmpty(scope)) {
    goto error;
  }
  ap = Ast_Node(AST_COMP);
  Ast_AddChild(ap, Ast_Snd());
  it = Link(scope);
  do {
    if (strcmp(token, it->value) == 0) {
      return ap;
    } else {
      Ast_AddChild(ap, Ast_Fst());
    }
  } while ((it = Link(it)) != Link(scope));
error:
  fprintf(stderr, "Unbound variable: %s.\n", token);
  THROW;
}
```

Numeric constants are simply returned quoted.

78a ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁77b 78b▷

```
static ast_t *
ParseNum(const char *cp)
{
  int total = 0;

  assert(isdigit(*cp));

  do {
    assert(isdigit(*cp));
    total = (10 * total) + (*cp - '0');
  } while (*++cp != '\0');
  return Ast_Quote(total);
}
```

Figure 4.2 shows how to parse a sum `(+ M1 ... Mn)`, where `M1`, ..., `Mn` are themselves expressions, proceeding by induction on n.

78b ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁78a 79▷

```
static ast_t *
ParseSum(lexer_t * const lexer, const symbol_t *scope)
{
  ast_t * root;

  assert(lexer);
  assert(lexer->type == LEX_PLUS);

  Consume(lexer);
  root = ParseExpr(lexer, scope);
  Consume(lexer);
  do {
    root = Ast_Pair(root, ParseExpr(lexer, scope));
    root = Ast_Pair(Ast_Plus(), root);
    root = Ast_Comp(2, root, Ast_App());
    Consume(lexer);
  } while (lexer->type != LEX_RBRACK);

  return root;
}
```

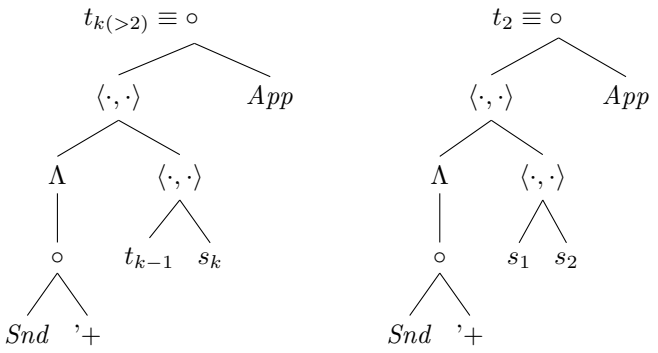

Figure 4.2: Mapping the input `(+ M1 ... Mn)` to an AST t_n, given trees $s_1, \ldots, s_n$ for $M_1, \ldots, M_n$.

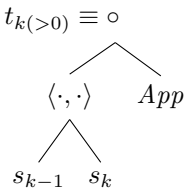

Figure 4.3: Mapping the input `(M0 M1 ... Mn)` to an AST t_n, given trees $s_0, \ldots, s_n$ for $M_1, \ldots, M_n$ and where M_0 is an abstraction.

In parsing an application whose operand is an abstraction, we want to make sure that the number of variables bound by the latter matches the number of operands. As we rather want the method for parsing abstractions to return an AST, just like the other methods implementing grammar rules, we communicate the number of bound variables through an `int` pointer that appears as an output parameter of `ParseAbs`. Assuming, then, that the abstraction takes the form `(lambda (x1 ... xn) N)` for some term `N`, to be referred to by `M0`, Figure 4.3 then shows how to parse `(M0 M1 ... Mn)` into an AST, motivating the following code.

79 ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁78b 80a▷

```
static ast_t *
ParseApp(lexer_t * const lexer, const symbol_t *scope)
{
  ast_t * root;
  int     cnt;

  assert(lexer);
```

```
  root = ParseAbs(lexer, scope, &cnt);
  while (cnt-- > 0) {
    Consume(lexer);
    root = Ast_Pair(root, ParseExpr(lexer, scope));
    root = Ast_Comp(2, root, Ast_App());
  }
  Expect(lexer, LEX_RBRACK);
  return root;
}
```

The parsing of an abstraction `(lambda (x1 ... xn) B)` proceeds in three steps.

80a ⟨*parser.c function definitions* 74c⟩+≡ (73b) ◁79

```
static ast_t *
ParseAbs(lexer_t * const lexer, const symbol_t *scope, int *cnt)
{
  ast_t * ap;
  int     i;

  assert(lexer);

  Match(lexer, LEX_LBRACK);
  Expect(lexer, LEX_LAMBDA);
  Expect(lexer, LEX_LBRACK);

  ⟨parse variable list 80b⟩
  ⟨parse body 81a⟩
  ⟨construct AST 81b⟩

  return ap;
}
```

First, we push new symbols onto the scope as we read the parameter list `x1`, ..., `xn`.

80b ⟨*parse variable list* 80b⟩≡ (80a)

```
Expect(lexer, LEX_VAR);
PushNewSymbol(&scope, lexer->token);
Consume(lexer);
for (*cnt = 1; lexer->type != LEX_RBRACK; ++*cnt) {
  Match(lexer, LEX_VAR);
  PushNewSymbol(&scope, lexer->token);
  Consume(lexer);
}
```

Next, the body `N` of the abstraction is parsed using the extended scope, returning some AST.

81a ⟨*parse body* 81a⟩≡ (80a)

```
Consume(lexer);
ap = ParseExpr(lexer, scope);
Expect(lexer, LEX_RBRACK);
```

To obtain the AST for the abstraction as a whole, we add n Λ-nodes.

81b ⟨*construct AST* 81b⟩≡ (80a)

```
for (i = *cnt; i > 0; --i) {
  ap = Ast_Cur(ap);
  Pool_Free(&g_symbol_pool, Pop(&scope));
}
```

4.4 The REPL

We conclude our exposition with the REPL, orchestrating the entire application pipeline from parsing the input down to optimizing and evaluating the AST.

82a ⟨*main.c* 82a⟩≡

```
#include <assert.h>
#include <stdio.h>
#include <string.h>

#include "ast.h"
#include "cam.h"
#include "env.h"
#include "except.h"
#include "lexer.h"
#include "optim.h"
#include "parser.h"
#include "pool.h"

⟨main.c constants 84a⟩
⟨main.c global variables 84d⟩
⟨main.c function definitions 82b⟩
```

The REPL operates in a loop, on each iteration reading in a closed term from standard input on a separate line and passing it on to the parser.

82b ⟨*main.c function definitions* 82b⟩≡ (82a) 83d▷

```
static int
Evaluate(const char * const buff)
{
  ast_t * ap;
  cam_t   cam;
  lexer_t lexer;
  optim_t optim;
  int     result = -1;

  ⟨parse input as ap 82c⟩
  ⟨optimize ap 83a⟩
  ⟨evaluate ap into result 83b⟩
  ⟨cleanup and return result 83c⟩
}
```

To parse the input into an AST, it suffices to compose a lexer with a parser.

82c ⟨*parse input as* ap 82c⟩≡ (82b)

```
Lexer_Init(&lexer, buff);
ap = Parse(&lexer);
```

We next keep running optimization passes over the generated AST until no more transformations can be applied. In between each two passes, we make sure to clean up the old AST so as not to run out of memory.

83a ⟨*optimize* `ap` 83a⟩≡ (82b)
```
do {
  Optim_Init(&optim);
  Ast_Traverse(ap, (visit_t *)&optim);
  Ast_Free(&ap);
  ap = Pop(&optim.stack);
  assert(IsEmpty(optim.stack));
} while (optim.cnt != 0);
```

Finally, we evaluate the AST and extract an integer result.

83b ⟨*evaluate* `ap` *into* `result` 83b⟩≡ (82b)
```
Cam_Init(&cam);
Ast_Traverse(ap, (visit_t *)&cam);
assert(cam.env->type == ENV_INT);
result = cam.env->u.num;
```

To prevent memory leaks, we should free any environment nodes allocated during evaluation, as well as the generated AST itself.

83c ⟨*cleanup and return* `result` 83c⟩≡ (82b)
```
Cam_Free(&cam);
Ast_Free(&ap);
return result;
```

The entry point to our application contains the looped invocation of `Evaluate`.

83d ⟨*main.c function definitions* 82b⟩+≡ (82a) ◁82b
```
int
main()
{
  char    buff[BUFF_SZ];
  char *  cp;

  for (;;) {
    ⟨read line into buff 84b⟩
    ⟨handle special commands 84c⟩
    ⟨eval and print 84e⟩
  }
}
```

Note input lines are read into an internal buffer, whose size we restrict to 256.

84a ⟨*main.c constants* 84a⟩≡ (82a)
```
enum {
  BUFF_SZ = 256
};
```

To read a line, we keep reading characters until we see `'\n'` or the buffer is full.

84b ⟨*read line into* `buff` 84b⟩≡ (83d)
```
for (cp=buff; cp-buff<BUFF_SZ && (*cp=getchar())!='\n'; ++cp)
  ;
if (cp - buff == BUFF_SZ) {
  fprintf(stderr, "Input too long.\n");
  continue;
}
*cp = '\0';
```

Intending for the interactive usage of the REPL, we signify the end of the session using a special command, as opposed to using `EOF` for said purpose.

84c ⟨*handle special commands* 84c⟩≡ (83d)
```
if (strcmp("halt", buff) == 0) {
  return 0;
}
```

The invocation of `Evaluate` may throw exceptions, which we will catch at at the top of the loop.

84d ⟨*main.c global variables* 84d⟩≡ (82a)
```
jmp_buf *   g_handler;
```

Exceptions are handled by clearing all memory pools and continuing with the next loop iteration.

84e ⟨*eval and print* 84e⟩≡ (83d)
```
TRY
  printf("%d\n", Evaluate(buff));
CATCH
  Pool_Clear(&g_ast_pool);
  Pool_Clear(&g_env_pool);
  Pool_Clear(&g_symbol_pool);
END
```

Bibliography

[1] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In *Conference on Functional Programming Languages and Computer Architecture*, pages 50–64. Springer, 1985.

[2] Roy L Crole. *Categories for types*. Cambridge University Press, 1993.

[3] Pierre-Louis Curien. Typed categorical combinatory logic. In *Colloquium on Trees in Algebra and Programming*, pages 157–172. Springer, 1985.

[4] Haskell Brooks Curry, Robert Feys, and William Craig. Combinatory logic. 1972.

[5] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[6] Christopher W Fraser and David R Hanson. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] David R Hanson. *C interfaces and implementations: techniques for creating reusable software*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[8] Geneva ISO. Iso 14977. *Information technology Syntactic metalanguage Extended BNF, Norm*, 1996.

[9] Ralph Johnson, Erich Gamma, Richard Helm, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Boston, Massachusetts: Addison-Wesley*, 1995.

[10] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[11] Donald Ervin Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[12] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[13] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[14] Norman Ramsey. Literate-programming can be simple and extensible. *IEEE Software*, 1993.

[15] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.