nils m holm

zen style programming

preface

A program is a description of an abstract, general solution to a specific problem. It is typically written in a formal language called a programming language. The primary purpose of a program is to be understood by fellow human beings, thereby spreading knowledge. In order to achieve maximal readability, a programming language should have certain properties:

1. It should be small and uniform;
2. It should be free from ambiguity;
3. It should provide a high degree of abstraction;
4. It should be independent from concrete computer architectures.

The first points are no-brainers. If a language is too complex or has no uniform syntax and semantics, programmers will have to look up things in the manual perpetually instead of concentrating on the actual problem. If the language introduces ambiguity, people will eventually choose one possible outcome internally and start writing programs that depend on their imagination instead of facts.

A high degree of abstraction means that the language should provide means of dealing with recurring tasks gracefully and without having to reinvent the wheel over and over again. This may seem like a contradiction to 1., but this text will show that this does not have to be the case.

A programming language that is used to describe algorithms in a readable way must be fully architecture-neutral. As soon as the language depends on features of a particular machine, the first principle is violated, because the knowledge that is necessary to understand a program then includes the knowledge of the underlying architecture.

The first part of this book introduces the concept of functional programming, describes a language that fulfills all of the above requirements, and shows how to solve simple problems in it.

Once a language has been chosen, it can be used to formulate solutions to all kinds of logic problems. Programming is limited to solutions of *logic* problems, because it cannot anwser questions like ‘‘how can we learn to live in peace?’’ and ‘‘why is life worth living?’’. These are the limits of science as we know it. The class of problems that *can* be solved by programs is much smaller. It typically involves very clearly defined tasks like

– find permutations of a set;
– find factors of an integer;
– represent an infinite sequence;
– translate formal language A to language B;
– find a pattern in a sequence of characters;
– solve a system of assertions.

Of course these tasks have to be defined in much greater detail in order to be able to solve them programmatically. This is what the second part of the book is about: creating general solutions to logic problems.

The language that will be used in this book is a minimalistic variant of Scheme called zenlisp. Its only data types are symbols and ordered pairs. Nothing else is required to describe algorithms for solving all of the logic problems listed above. The language may appear useless to you, because it does not provide any access to ‘‘real-world’’ application program interfaces and it cannot be used to write interactive programs, but I have chosen this language on purpose: it demonstrates that programming does not depend on these features.

The second part of the book shows how to apply the techniques of the first part to some problems of varying complexity. The topics discussed in this part range from simple functions for sorting or permuting lists to regular expression matching, formal language translation, and declarative programming. It contains the full source code to a source-to-source compiler, a meta-circular interpreter and a logic programming system.

The third part, finally, shows how to implement the abstraction layer that is necessary for solving problems in an abstract way on a concrete computer. It reproduces the complete and heavily annotated source code for an interpreter of zenlisp.

The first chapter of this part strives to deliver an example of readable code in a language that is not suitable for abstraction. It attempts to develop a programming style that does not depend on annotations to make the intention of the programmer clear. Comments are interspersed between functions, though, because prose is still easier to read than imperative code.

At this point the tour ends. It starts with an abstract and purely symbolic view on programming, advances to the application of symbolic programming to more complex problems, and concludes with the implementation of a symbolic programming system on actual computer systems.

I hope that you will enjoy the tour!

Nils M Holm, September 2008

contents

part one symbolic programming

This part discusses the foundations of symbolic programming by means of a purely symbolic, lexically scoped, functional variant of LISP called zenlisp.

Zenlisp is similar to Scheme, but simpler.

Being purely symbolic, the language has only two fundamental data types: the symbol and the ordered pair.

Zenlisp implements the paradigm of *functional programming*. Functional programming focuses on the evaluation of expressions. Programs are sets of functions that map values to values. Tail-recursive expressions evaluate in constant space, making iteration a special case of recursion.

Zenlisp programs are typically free of side effects.

Although the techniques described in this text are presented in a LISPy language, the purely symbolic approach makes it easy to adapt those techniques to other functional languages and maybe even to languages of other paradigms.

1. basic aspects

1.1 symbols and variables

This is a *quoted symbol*:

```
'marmelade
```

It is called a *quoted* symbol because of the quote character in front of it.

Anything that is quoted *reduces to* itself:

```
'marmelade => 'marmelade
```

The **=>** operator reads ‘‘*reduces to*’’ or ‘‘*evaluates to*’’.

The lefthand side of **=>** is a program and its righthand is called the *normal form* (or the ‘‘result’’) of that program.

A symbol that is *not* quoted is called a *variable*.

Here is a variable:

```
food
```

The normal form of a variable is the value associated with that variable:

```
(define food 'marmelade)
food => 'marmelade
```

The above **`define`** *binds* the value `'marmelade` to the variable *food*.

After binding a variable to a value, each reference to the variable results in the value bound to that variable; the variable reduces to its value:

```
food => 'marmelade
```

Symbols and variables are independent from each other:

```
(define marmelade 'fine-cut-orange)
marmelade => 'fine-cut-orange
```

While *marmelade* now refers to `'fine-cut-orange`, the quoted symbol `'marmelade` still reduces to itself and *food* still reduces to `'marmelade`:

```
'marmelade => 'marmelade
      food => 'marmelade
```

Once `defined`, the value of a variable normally does not change.

A symbol that has no value associated with it is not a valid variable:

```
undefined-symbol => bottom
```

Bottom denotes an *undefined value*.

Anything that reduces to bottom should be considered an error.

A quoted symbol is its own value, so a quoted symbol without an association is fine:

```
'undefined-symbol => 'undefined-symbol
```

1.2 functions

The following *expression* applies the **`append`** *function* to two *arguments*:

```
(append '(pizza with) '(extra cheese))
=> '(pizza with extra cheese)
```

The *data* `'(pizza with)` and `'(extra cheese)` are *lists*. **`Append`** appends them.

Lists are in their normal forms because they are quoted:

```
'(pizza with cheese) => '(pizza with cheese)
```

The normal form of a function application is the value returned by the applied function:

```
(reverse '(pizza with pepperonies))
=> '(pepperonies with pizza)
```

Lists and function applications share the same notation; they differ only by the quote character that is attached to lists:

```
 (reverse '(ice water)) => '(water ice)
'(reverse '(ice water)) => '(reverse '(ice water))
```

Attaching a quote character to a function application turns it into a list, but...

Removing a quote from a list does not necessarily turn it into a function application:

```
(pizza with pepperonies) => bottom
```

This does not work, because no function with the name ‘‘pizza’’ has been defined before.

You can define one, though:

```
(define (pizza topping)
  (list 'pizza 'with topping))
```

defines a function named *pizza* that takes one argument named *topping*.

The *body* of the function is an application of **`list`** to three arguments: the symbols `'pizza` and `'with` and the variable *topping*. The body of a function is sometimes also called the *term* of that function.

When *pizza* is applied, **`list`** forms a new list containing the given arguments:

```
(pizza 'anchovies) => '(pizza with anchovies)
```

BTW, the quotes of the symbols `'pizza` and `'with` have not vanished.

A quote character in front of a list quotes everything contained in that list:

```
   '(gimme a (pizza 'hot-chili))
=> '(gimme a (pizza 'hot-chili))
```

If you want to reduce the *members* of a list, you have to use the **list** function:

```
(list 'gimme 'a (pizza 'hot-chili))
=> '(gimme a (pizza with hot-chili))
```

1.2.1 calling functions

Pizza-2 is like *pizza*, but accepts two arguments:

```
(define (pizza-2 top1 top2)
  (list 'pizza 'with top1 'and top2))
```

The variables *top1* and *top2* are the *variables* of the function. They are sometimes also called its *formal arguments.*

The values the function is applied to are called *actual arguments* or just *arguments*. A list of variables of a function is called its *argument list*.

In the following example, `'(extra cheese)` and `'(hot chili)` are (actual) arguments:

```
(pizza-2 '(extra cheese) '(hot chili))
=> '(pizza with (extra cheese) and (hot chili))
```

Here is what happens during the above function application:

1. the values of *top1* and *top2* are saved;
2. `'(extra cheese)` is bound to *top1*;
3. `'(hot chili)` is bound to *top2*;
4. `(list 'pizza 'with top1 'and top2)` is reduced, giving a result *R*;
5. *top1* and *top2* are bound to the values saved in 1.;
6. *R* is returned.

Because of 1. and 5., the variables of a function are *local* to that function.

Therefore multiple functions may share the same variable names. Each of theses function has its own local *x*:

```
(define (f x) (append x x))
(define (g x) (reverse x))
```

Arguments of functions are matched by position:

```
(pizza-2 'olives 'pepper) => '(pizza with olives and pepper)
(pizza-2 'pepper 'olives) => '(pizza with pepper and olives)
```

1.2.2 function composition

This is a composition of the functions **`reverse`** and **`append`**:

```
(reverse (append '(ice with) '(juice orange)))
=> '(orange juice with ice)
```

Function composition is used to form new functions from already existing ones:

```
(define (palindrome x)
  (append x (reverse x)))
```

In *palindrome*, the second argument of **`append`** is the value returned by **`reverse`**.

Zenlisp uses *applicative order evaluation*.

Therefore `(reverse x)` is first reduced to its normal form and that normal form is passed as an argument to **`append`**.

Multiple arguments to the same function are not reduced in any specific order, but in the examples left-to-right evaluation is assumed.

Here is a sample application of *palindrome*:

```
(palindrome '#12345)
```

The datum `'#12345` is just a zenlisp short cut for `'(1 2 3 4 5)` — every list of single-character symbols can be abbreviated this way.

The application reduces as follows (the `->` operator denotes a *partial reduction*):

```
(palindrome '#12345)
-> (append x (reverse x))             ; reduced palindrome
-> (append '#12345 (reverse x))       ; reduced first x
-> (append '#12345 (reverse '#12345)) ; reduced second x
-> (append '#12345 '#54321)           ; reduced application of reverse
=> '#1234554321                       ; reduced application of append
```

In a partial reduction, one or multiple sub-expressions of a function application (like variables or embedded applications) are reduced to their normal forms.

1.3 conditions

The symbol **`:f`** denotes logical falsity:

```
:f => :f
```

`:F` reduces to itself, so it does not have to be quoted.

The symbols **`:t`** and **`t`** denote logical truth:

```
:t => :t
 t => :t
```

`:T` and **`t`** both reduce to **`:t`**, so they do not have to be quoted either.

In case you wonder why there are *two* values representing truth, **`t`** simply looks better than **`:t`** in some contexts.

The **`cond`** *pseudo function* implements *conditional reduction*:

```
(cond (:f 'bread)
      (:t 'butter))
=> 'butter
```

Each argument of **`cond`** is called a *clause*.

Each clause consists of a *predicate* and a *body*:

(*predicate body*)

`Cond` works as follows:

1. it reduces the predicate of its first clause to its normal form;
2. if the predicate reduces to truth, the value of **`cond`** is the normal form of the body associated with that predicate;
3. if the predicate reduces to falsity, **`cond`** proceeds with the next clause.

`Cond` returns as soon as a true predicate is found:

```
(cond (:f 'bread)
      (:t 'butter)
      (:t 'marmelade))
=> 'butter
```

Therefore, the above **`cond`** will *never* return `'marmelade`.

It is an error for **`cond`** to run out of clauses:

```
(cond (:f 'false)
      (:f 'also-false))
=> bottom
```

Therefore, the last clause of **`cond`** should always have constant truth as its predicate, so it can catch all remaining cases.

Here is a function that uses **`cond`**:

```
(define (true value)
  (cond (value (list value 'is 'true))
        (t (list value 'is 'false))))
```

True finds out whether a value is "true" or not.

Let us try some values:

```
(true :f) => '(:f is false)
```

As expected.

```
(true :t) => '(:t is true)
```

Also fine. Here are the truth values of some other expressions: [1]

```
(true 'orange-fine-cut) => '(orange-fine-cut is true)
              (true '0) => '(0 is true)
            (true true) => '({closure (value)} is true)
              (true ()) => '(() is true)
```

It seems that most values thrown at *true* turn out to be true.

Indeed:

`Cond` interprets all values except for `:f` as truth. Only `:f` is false.

1 We will investigate at a later time why *true* reduces to `{closure (value)}`.

1.4 recursion

Here are some interesting functions:

`Car` extracts the first member of a list:

```
(car '(first second third)) => 'first
            (car '#abcdef) => 'a
```

The **`cdr`** function extracts the tail of a list:

```
(cdr '(first second third)) => '(second third)
            (cdr '#abcdef) => '#bcdef
```

The tail of a list of one member is **`()`**:

```
(cdr '(first)) => ()
```

`()` is pronounced “nil”; it represents the *empty list*.

The **`null`** *predicate* tests whether its argument is **`()`**:

```
(null '#abcde) => :f
(null ()) => :t
```

A function is called a predicate when it *always* returns either **`:t`** or **`:f`**.

The **`eq`** predicate tests whether two symbols are identical:

```
(eq 'orange 'orange) => :t
(eq 'orange 'apple)  => :f
```

`Memq` makes use of all of the above functions:

```
(define (memq x a)
  (cond ((null a) :f)
        ((eq x (car a)) a)
        (t (memq x (cdr a)))))
```

It locates the first occurrence of *x* in the list of symbols *a*:

```
(memq 'c '#abcde) => '#cde
```

When *a* does not contain *x*, **`memq`** returns **`:f`**:

```
(memq 'x '#abcde) => :f
```

When the list passed to **`memq`** is empty, the first clause of the **`cond`** of **`memq`** applies and **`memq`** returns **`:f`**:

```
(memq 'x ()) => :f
```

The second clause uses **`eq`** to find out whether *x* and `(car a)` denote the same symbol. If so,

memq returns *a*:

```
(memq 'x '#x) => '#x
```

The last clause applies **memq** to the *tail* of the list *a*:

```
(memq x (cdr a))
```

Because **memq** applies **memq** in order to compute its own result, it is said to *recurse*; **memq** is called a *recursive function*.

An application of **memq** reduces as follows:

```
(memq 'c '#abcde)
-> (cond ((null '#abcde) :f)
         ((eq 'c (car '#abcde)) '#abcde)
         (t (memq 'c (cdr '#abcde))))
-> (cond ((eq 'c (car '#abcde)) '#abcde)
         (t (memq 'c (cdr '#abcde))))
-> (cond (t (memq 'c (cdr '#abcde))))
-> (memq 'c (cdr '#abcde))
-> (memq 'c '#bcde)
-> (memq 'c '#cde)
=> '#cde
```

Each clause of **cond** covers one *case*.

The non-recursive cases of a recursive function are called its *trivial cases*, the recursive cases are called its *general cases*.

Recursion is used to express iteration.

1.5 forms and expressions

Car and **cdr** extract the *head* (the first member) and the *tail* (all members but the first) of a list:

```
(car '(large banana split)) => 'large
(cdr '(large banana split)) => '(banana split)
```

The **cons** function creates a fresh list by attaching a new head to an existing list:

```
(cons 'banana ()) => '(banana)
(cons 'banana '(split)) => '(banana split)
```

However, the second argument of **cons** does not have to be a list:

```
(cons 'heads 'tails) => '(heads . tails)
```

A structure of the form

```
(car-part . cdr-part)
```

is called a *dotted pair*.

The functions **cons**, **car**, and **cdr** are correlated in such a way that

```
(cons (car x) (cdr x))  =  x
```

holds for any pair *x*.

The car part and cdr part of each pair may be another pair:

```
((caar . cdar) . (cdar . cddr))
```

The name "caar" denotes the "car part of a car part", "cdar" denotes the "cdr part of a car part", etc.

There is a set of equally named functions that extract these parts from nested pairs:

```
(caar '((caar . cdar) . (cadr . cddr))) => 'caar
(cdar '((caar . cdar) . (cadr . cddr))) => 'cdar
(cadr '((caar . cdar) . (cadr . cddr))) => 'cadr
(cddr '((caar . cdar) . (cadr . cddr))) => 'cddr
```

Zenlisp provides functions to extract data from up to four levels of nested pairs.

For instance **cddddr** extracts the cdr^4 part (the tail starting at the fourth member of a list) and **caddr** returns the "car of the cdr of the cdr" of a datum (which happens to be the second member of a list):

```
(cddddr '#abcdef) => '#ef
 (caddr '#abcdef) => 'c
```

Hint: To decode functions for accessing nested pairs, read the a's and d's in their names backward:

```
(cadadr '(a (b c) d)) ; remove last 'd', do cdr
-> (cadar '((b c) d)) ; remove last 'a', do car
-> (cadr '(b c))      ; remove last 'd', do cdr
-> (car '(c))         ; do final car
=> 'c
```

1.5.1 lists

A *list* is

1. either the empty list **()**;
2. or a pair whose cdr part is a *list*.

The **listp** predicate tests whether a datum is a list: [2]

2 Appending a "p" to the name of a predicate is ancient LISP tradition, but zenlisp follows this tradition in a rather liberal way.

```
(define (listp x)
  (cond ((null x) :t)
        ((atom x) :f)
        (t (listp (cdr x)))))
```

`Listp` uses the **`atom`** predicate, which tests whether its argument is *atomic*. Anything that cannot be split (by **`car`** or **`cdr`**) is atomic:

```
(car 'symbol) => bottom
     (car ()) => bottom
(cdr 'symbol) => bottom
     (cdr ()) => bottom
```

Let us apply **`listp`** to some data:

```
             (listp ()) => :t
(listp '(banana split)) => :t
        (listp '#abcde) => :t
```

Fine, and now some negatives:

```
         (listp 'banana) => :f
(listp '(heads . tails)) => :f
```

`'Banana` is obviously not a list, nor is the tail of `'(heads . tails)`.

How about these:

```
  (listp '(define (f x) x)) => :t
(listp '(x (((y . z))) ())) => :t
```

Yes, lists may contain pairs and lists, and they may be nested to any level.

And this?

```
(listp (append '#abcde 'x)) => :f
```

Whatever appending a symbol to a list gives, it is not a list:

```
(append '#abcde 'x) => '(a b c d e . x)
```

The resulting structure is a hybrid of a dotted pair and a list; it is called a *dotted list*. (Sometimes it is also called an *improper list*, because “proper” lists end with **`()`**.)

Because lists are pairs, the functions **`caar`**...**`cddddr`** can be used to extract members of lists, too.

For instance:

```
  (caar '((first) (second) (third))) => 'first
  (cadr '((first) (second) (third))) => '(second)
  (cdar '((first) (second) (third))) => '()
  (cddr '((first) (second) (third))) => '((third))
```

```
 (caddr '((first) (second) (third))) => '(third)
 (cdddr '((first) (second) (third))) => ()
(caaddr '((first) (second) (third))) => 'third
```

1.5.2 forms

Each form is either a symbol or a pair.

These are forms:

```
marmelade
cdr
:f
(heads . tails)
(banana ; this is a comment
        split)
(define (f x) (f (f x)))
(1 2 3 4 . 5)
#hello-world
```

A *comment* can be placed anywhere inside of a form (but *not* inside of a symbol name!) using a semicolon.

A comment extends up to the end of the current line.

For some forms, there are different notations.

Lists may be expanded to pairs:

```
(large banana split)  =  (large . (banana . (split . ())))
```

There is no really good reason to do so, but it shows that

Every list is a pair. (But not every pair is a list.)

Lists that consist of single-character symbols exclusively may be *condensed*:

```
(h e l l o - w o r l d !)  =  #hello-world!
```

Condensed forms are useful because they are easier to comprehend, easier to type, and save space.

Did you notice that no form in this subsection was *quoted*?

This is because "form" is an abstract term.

A form is turned into a *datum* by applying the **`quote`** pseudo function to it:

```
(quote (banana split)) => '(banana split)
```

But you do not have to use **`quote`** each time you want to create a datum, because

```
'(banana split) => '(banana split)
```

1.5.3 expressions

An expression is a form with a meaning.

In zenlisp, there is no difference between *expressions* and *programs*.
Every expression is a program and every program consists of one or multiple expressions.

```
(car '(fruit salad))
```

is a program that extracts the first member of a specific list.

```
(define (palindrome x)
  (append x (reverse x)))
```

is an expression with a *side effect*.

A side effect causes some state to change.

The side effect of **`define`** is to create a *global definition* by binding a variable to a value. In the above case, the value is a function.

The definition is called ''global'', because its binding does not occur inside of a specific function.

`Define` does have a result, but it is mostly ignored:

```
(define (pizza x) (list 'pizza 'with x)) => 'pizza
```

Because **`define`** is called only for its side effect, it is called a *pseudo function* or a *keyword*.

Another property of pseudo functions is that they are called *by name*, i.e. their arguments are not reduced before the function is applied.

This is why the clauses of **`cond`** and the arguments of **`quote`** and **`define`** do not have to be quoted:

```
(cond (:f 'pizza) (t 'sushi))
(quote (orange juice))
(define (f x) (f f x))
```

Each expression is either a variable or a (pseudo) function application.

1.6 recursion over lists

Here is **`reverse`**:

```
(define (reverse a)
  (cond ((null a) ())
        (t (append (reverse (cdr a))
                   (list (car a))))))
```

Reverse reverses a list.

The trivial case handles the empty list, which does not have to be reversed at all.

The general case reverses the *cdr part* (the rest) of the list and then appends a list containing the *car part* (first member) of the original list to the result.

This is how **reverse** works:

```
(reverse '#abc)
-> (append (reverse '#bc) (list 'a))
-> (append (append (reverse '#c) (list 'b)) (list 'a))
-> (append (append (append (reverse ()) (list 'c)) (list 'b))
           (list 'a))
-> (append (append (append () (list 'c)) (list 'b)) (list 'a))
-> (append (append '#c (list 'b)) (list 'a))
-> (append '#cb (list 'a))
=> '#cba
```

Each member of the argument of **reverse** adds one application of **append**.

This is called *linear recursion.*

In many cases, linear recursion can be avoided by adding an additional argument that carries an intermediate result.

Reverse can be modified in such a way:

```
(define (reverse2 a r)
  (cond ((null a) r)
        (t (reverse2 (cdr a)
                     (cons (car a) r)))))

(define (reverse a) (reverse2 a ()))
```

Reverse is now a “wrapper” around *reverse2*, and *reverse2* does the actual work:

```
(reverse2 '#abc ())
-> (reverse2 '#bc '#a)
-> (reverse2 '#c '#ba)
-> (reverse2 () '#cba)
=> '#cba
```

Because the intermediate result does not grow during the reduction of *reverse2*, the function is said to *reduce in constant space.*

This is achieved by rewriting the recursive application of **reverse** as a *tail call.*

A function *call* is the same as a function application.

A tail call is a function call in a *tail position.*

In the expression

```
(append (reverse (cdr a)) (list (car a)))
```

`reverse` is *not* in a tail position, because **`append`** is called when **`reverse`** returns.

In the expression

```
(reverse2 (cdr a) (cons (car a) r))
```

reverse2 is in a tail position, because *reverse2* is the last function called in the expression.

BTW, **`cond`** does not count, because

```
(cond (t (reverse2 (cdr a) (cons (car a) r))))
```

can be rewritten as

```
(reverse2 (cdr a) (cons (car a) r))
```

So:

1. the outermost function in a function body is in a tail position;
2. the outermost function in a **`cond`** body is in a tail position.

A function that uses recursion in tail positions exclusively is called a *tail-recursive* function.

Tail recursion is more efficient than linear recursion.

Here is another recursive function, *append2*:

```
(define (append2 a b)
  (cond ((null a) b)
        (t (cons (car a)
                 (append2 (cdr a) b)))))
```

It is called *append2*, because it accepts two arguments. **`Append`** accepts any number of them:

```
(append '#he '#llo '#- '#wor '#ld) => '#hello-world
```

But this is not the worst thing about *append2*.

Append2 conses **`(car a)`** to the result of an application of *append2*, so it is not tail-recursive.

Can you write a tail-recursive version of *append2*?

Here it is:

```
(define (r-append2 a b)
  (cond ((null a) b)
        (t (r-append2 (cdr a)
                      (cons (car a) b)))))
```

```
(define (append2 a b)
  (r-append2 (reverse a) b))
```

And this is how it works:

```
(append2 '#abc '#def)
-> (r-append (reverse '#abc) '#def)
-> (r-append '#cba '#def)
-> (r-append '#ba '#cdef)
-> (r-append '#a '#bcdef)
-> (r-append () '#abcdef)
=> '#abcdef
```

Are there any functions that cannot be converted to tail-recursive functions?

Yes, there are. You will see some of them in the following chapter.

2. more interesting aspects

2.1 variadic functions

Here is *intersection*:

```
(define (intersection a b)
  (cond ((null a) ())
        ((memq (car a) b)
          (cons (car a)
                (intersection (cdr a) b)))
        (t (intersection (cdr a) b))))
```

It computes the intersection of two sets of symbols:

```
(intersection '#abcd '#cdef) => '#cd
(intersection '#abcd '#wxyz) => ()
```

If you want to form the intersection of more than two sets, you have to compose applications of *intersection*:

```
(define (intersection3 a b c)
  (intersection a (intersection b c)))
```

To process a variable number of sets, you can pass the sets to a function inside of a list. This is how *intersection-list* works:

```
(define (intersection-list a*)
  (cond ((null a*) a*)
        ((null (cdr a*)) (car a*))
        (t (intersection (car a*)
                         (intersection-list (cdr a*))))))
```

Intersection-list forms the intersection of all sets contained in a list:

```
(intersection-list '()) => ()
(intersection-list '(#abcd)) => '#abcd
(intersection-list '(#abcd #bcde)) => '#bcd
(intersection-list '(#abcd #bcde #cdef)) => '#cd
```

Here is the code of **`list`**:

```
(define (list . x) x)
```

Yes, that's all. Really.

`List` is a *variadic function*, a function that accepts a variable number of arguments.

The dot in front of the variable (*x*) of **`list`** says: "bind a list containing all actual arguments to that variable":

```
                  (list 'orange) => '(orange)
           (list 'orange 'juice) => '(orange juice)
(list 'orange 'juice 'with 'ice) => '(orange juice with ice)
```

Except for being variadic, **`list`** is an ordinary function.

Because arguments are reduced to their normal forms *before* they are passed to **`list`**, lists can include dynamic values:

```
(list (cons 'heads 'tails) (intersection '#abcde '#cdefg))
=> '((heads . tails) #cde)
```

When no arguments are passed to **`list`**, *x* is bound to a list of no arguments:

```
(list) => ()
```

Here is a version of *intersection-list* that accepts a variable number of sets instead of a list of sets:

```
(define (intersection* . a*)
  (cond ((null a*) a*)
        ((null (cdr a*)) (car a*))
        (t (intersection
             (car a*)
             (apply intersection* (cdr a*))))))
```

There are two differences between *intersection-list* and *intersection**:

1. *intersection** takes a variable number or arguments;
2. *intersection** uses **`apply`** to recurse.

`Apply` applies a function to a list of arguments:

(apply fun (list arg_1 ... arg_n)) = (fun arg_1 ... arg_n)

Here are some examples:

```
            (apply cons '(heads tails)) => '(heads . tails)
      (apply intersection* '(#abc #bcd)) => '#bc
(apply intersection* (cdr '(#abc #bcd))) => '#bcd
```

`Apply` can be used to apply a function to a dynamically generated list of arguments.

In *intersection**, it applies the function to the tail of the argument list:

```
(intersection* '#abc '#bcd '#cde)
-> (intersection '#abc (apply intersection*
                              (cdr '(#abc #bcd #cde))))
-> (intersection '#abc (apply intersection* '(#bcd #cde)))
-> (intersection '#abc (intersection '#bcd
                                     (apply intersection* '(#cde))))
-> (intersection '#abc (intersection '#bcd '#cde))
-> (intersection '#abc '#cd)
=> '#c
```

BTW, **`apply`** can safely be used in tail calls.

Can you write a tail-recursive version of *intersection**?

No solution is provided at this point.

This function creates a non-empty list:

```
(define (non-empty-list first . rest)
  (cons first rest))
```

When applied to some actual arguments, it behaves in the same way as **`list`**:

```
(non-empty-list 'a 'b 'c) => '#abc
   (non-empty-list 'a 'b) => '#ab
      (non-empty-list 'a) => '#a
```

Applying it to no arguments at all is undefined, though:

```
(non-empty-list) => bottom
```

This is because *non-empty-list* expects *at least one* argument.

There must be one argument for each variable in front of the dot of its dotted argument list:

```
(non-empty-list first . rest)
```

First is bound to the first argument and *rest* is bound to the list of “remaining” argument, if any.

If there is exactly one argument, *rest* is bound to **`()`**.

There may be any number of arguments in front of the dot:

```
(define (skip-3 dont-care never-mind ignore-me . get-this) get-this)
```

2.2 equality and identity

All symbols are unique.

Therefore all symbol names that are equal refer to the same symbol:

```
marmelade  marmelade  marmelade  marmelade  marmelade
```

All these names refer to the same symbol named “marmelade”.

Two instances of the same symbol are called *identical*.

Identity is expressed using the **`eq`** predicate:

```
(eq 'marmelade 'marmelade) => :t
(eq 'fine-cut 'medium-cut) => :f
```

There is only one *empty list* in zenlisp, so all instances of **()** are identical, too:

```
(eq () ()) => :t
```

But **eq** can do more than this:

```
 (eq 'symbol '(a . pair)) => :f
(eq 'symbol '(some list)) => :f
```

When one argument of **eq** is neither a symbol nor **()** and the other one is either a symbol or **()**, **eq** is guaranteed to reduce to falsity.

This may be considered as a means of "built-in type checking".

So two forms are identical, if...

– they denote the same symbol;
– they are both **()**.

Two forms are *not* identical, if one of them is a pair and the other is not a pair.

When both arguments of **eq** are pairs, the result is *undefined*:

```
(eq '(a . b) '(a . b)) => bottom
(eq '#abcdef '#abcdef) => bottom
```

Note that "undefined" in this case means that the result is totally unpredictable. The application of **eq** to two equal-looking pairs can yield **:t** at one time and **:f** at another.

Therefore:

Never apply eq to two pairs.

2.2.1 comparing more complex structures

These two lists may or may not be identical:

```
'(bread with butter and marmelade)
'(bread with butter and marmelade)
```

But a quick glance is enough to find out that both lists contain identical symbols at equal positions, so they could be considered *equal*.

What about these lists:

```
'(bread with (butter and marmelade))
'((bread with butter) and marmelade)
```

Although the lists contain identical symbols, you would certainly not consider them equal, because they contain different sublists.

Here is a better approach:

Two forms are equal if...

– they are both the same symbol;
– they are both **`()`**;
– they are both pairs and contain equal car and cdr parts.

The **`equal`** function tests whether two forms are equal:

```
(define (equal a b)
  (cond ((atom a) (eq a b))
        ((atom b) (eq a b))
        ((equal (car a) (car b))
          (equal (cdr a) (cdr b)))
        (t :f)))
```

`Equal` returns truth whenever **`eq`** returns truth:

```
(equal 'fine-cut 'fine-cut) => :t
              (equal () ()) => :t
```

In addition, it returns truth when applied to two pairs that *look equal*:

```
(equal '(bread (with) butter)
       '(bread (with) butter)) => :t
```

Because it recurses into the car and cdr parts of its arguments, it detects differences even in nested lists:

```
(equal '(bread (with) butter)
       '(bread (without) butter)) => :f
```

Because **`equal`** makes sure that **`eq`** is only applied to arguments that do not cause undefined results, it can be applied safely to any type of datum.

Use `eq` to express identity and `equal` to express equality.

Here is **`member`**:

```
(define (member x a)
  (cond ((null a) :f)
        ((equal x (car a)) a)
        (t (member x (cdr a)))))
```

`Member` is similar to **`memq`** [page 15], but uses **`equal`** instead of **`eq`**.

Therefore it can find members that **`memq`** cannot find:

```
  (memq '(with) '(bread (with) butter)) => bottom
(member '(with) '(bread (with) butter)) => '((with) butter)
```

2.3 more control

This is how the **or** pseudo function works:

```
(or 'sushi 'pizza 'taco) => 'sushi
     (or :f :f 'taco :f) => 'taco
            (or :f :f :f) => :f
```

It returns the normal form of the first one of its arguments that does not reduce to falsity, or falsity if all of its arguments reduce to falsity.

Or can be expressed using **cond**:

```
(or a b)     =  (cond (a a) (t b))
(or a b c)   =  (cond (a a) (b b) (t c))
```

From this equivalence follows that

```
(or a)   =  (cond (t a))   =  a
```

In addition, applying **or** to zero arguments yields the neutral element of the logical "or":

```
(or) => :f
```

This is how the **and** pseudo function works:

```
(and 'tomato 'lettuce 'bacon) => 'bacon
      (and 'tomato :f 'bacon) => :f
```

It returns the first one of its arguments that does not reduce to truth, or the normal form of its last argument if all of them reduce to truth.

And can be expressed using **cond**:

```
(and a b)     =  (cond (a b) (t :f))
(and a b c)   =  (cond (a (cond (b c)
                               (t :f)))
                       (t :f))
```

In addition:

```
(and a)   =  a
```

Applying **and** to zero arguments yields the neutral element of the logical "and":

```
(and) => :t
```

And and **or** implement so-called *short circuit boolean reduction.*

Both of them stop evaluating their arguments as soon as they find a true or false value respectively:

```
(and :f (bottom)) => :f
 (or :t (bottom)) => :t
```

Bottom is a function that evaluates to bottom. Its result is undefined for any arguments passed to it:

```
                 (bottom) => bottom
             (bottom 'foo) => bottom
(bottom 'foo (list 'bar)) => bottom
```

Because of a principle known as *bottom preservation*, each function that takes a bottom argument must itself reduce to bottom:

```
 (atom (bottom)) => bottom
(eq 'x (bottom)) => bottom
(pizza (bottom)) => bottom
```

However, **and** and **or** are *pseudo functions* and their arguments are passed to them in *unreduced* form.

Because

– **and** never reduces any arguments following a ‘‘false’’ one
and
– **or** never reduces any arguments following a ‘‘true’’ one,

bottom preservation does not apply, and so:

```
(and :f (bottom)) => :f
 (or :t (bottom)) => :t
```

Bottom preservation does not apply to and, or, and cond.

By using **and** and **or**, two clauses of **equal** [page 28] can be saved:

```
(define (equal a b)
  (cond ((or (atom a) (atom b))
          (eq a b))
        (t (and (equal (car a) (car b))
                (equal (cdr a) (cdr b))))))
```

2.4 structural recursion

Here is the *replace* function:

```
(define (replace old new form)
  (cond ((equal old form) new)
        ((atom form) form)
        (t (cons (replace old new (car form))
                 (replace old new (cdr form))))))
```

Replace replaces each occurrence of *old* in *form* with *new*:

```
            (replace 'b 'x '#aabbcc) => '#aaxxcc
(replace 'old 'new '(old (old) old)) => '(new (new) new)
```

It can substitute data of any complexity in data of any complexity:

```
(replace '(g x) '(h x) '(f (g x))) => '(f (h x))
```

To do so, it uses a kind of recursion that is even more expensive than linear recursion.

Remember linear recursion? It is what happens when a function has to wait for a recursive call to complete.

Append2 (which first occurred on page 22) is linear recursive, because **cons** must wait until the recusive call to *append2* returns:

```
(define (append2 a b)
  (cond ((null a) b)
        (t (cons (car a)
                 (append2 (cdr a) b)))))
```

In *replace*, **cons** must wait for the completion of *two* recursive calls.

Therefore, the space required for intermediate results of *replace* grows even faster than the space required by linear recursive functions:

```
(replace 'b 'x '((a . b) . (c . d)))
-> (cons (replace 'b 'x '(a . b))
         (replace 'b 'x '(c . d)))
-> (cons (cons (replace 'b 'x 'a)
               (replace 'b 'x 'b))
         (cons (replace 'b 'x 'c)
               (replace 'b 'x 'd)))
```

In the worst case *replace* adds two applications of itself each time it recurses. It has to do so, because all atoms of a pair have to be visited before *replace* can return. In this case, the space required by the function grows exponentially.

Because this kind of recursion is required to process recursive structures, it is called *structural recursion.*

When the structure to be traversed is "flat" — like a list — structural recursion is no more expensive than linear recursion:

```
(replace 'c 'x '#abc)
-> (cons (replace 'c 'x 'a)
         (replace 'c 'x '#bc))
-> (cons (replace 'c 'x 'a)
         (cons (replace 'c 'x 'b)
               (replace 'c 'x '#c)))
```

```
-> (cons (replace 'c 'x 'a)
         (cons (replace 'c 'x 'b)
               (cons (replace 'c 'x 'c) ())))
```

Is there anything that can be done about structural recursion?

In the case of *replace*: no.

As the name already suggests:

Structural recursion is needed to traverse recursive structures.

There are not many occasions that require structural recursion, though.

The *contains* function traverses a recursive structure, but without combining the results of its recursive calls:

```
(define (contains x y)
  (cond ((equal x y) :t)
        ((atom y) :f)
        (t (or (contains x (car y))
               (contains x (cdr y))))))
```

Contains is similar to *replace*: its trivial cases return atoms, and its general case recurses twice.

However, the second recursive call to *contains* is a tail call.

Why?

When the first recursive call to *contains* returns truth, **`or`** does not even perform the second call and returns truth immediately.

When the first recursive call returns falsity, **`:f`** can be substituted for

```
(contains x (car y))
```

in

```
(or (contains x (car y))
    (contains x (cdr y)))
```

which leads to

```
(or :f (contains x (cdr y)))
```

and because

```
(or :f x)  =  (or x)  =  x
```

the second call to *contains* must *always* be a tail call.

The **`or`** pseudo function only waits for the first recursive call, which makes *contains* effectively linear recursive.

The first recursive call cannot be eliminated, because it reflects an inherent property of the structure to be traversed.

For the same reason, **equal** [page 30] is linear recursive.

2.5 functions revisited

This is an *anonymous function*:

```
(lambda (topping) (list 'pizza 'with topping))
```

It is equivalent to the named function *pizza* [page 11]:

```
((lambda (topping) (list 'pizza 'with topping))
 'pineapple)
=> '(pizza with pineapple)
```

Anonymous functions are created by the pseudo function **lambda**:

```
(lambda (topping) (list 'pizza 'with topping))
=> {closure (topping)}
```

Lambda creates a *closure* from the anonymous function.

Forms delimited by curly braces are *unreadable*:

```
{no matter what} => bottom
```

They are used to represent data that have no unambiguous external representation.

All zenlisp functions are either *primitive functions* or closures.

Define, **cdr**, and **lambda** itself are primitive functions:

```
define => {internal define}
   cdr => {internal cdr}
lambda => {internal lambda}
```

Other pre-defined zenlisp functions are closures:

```
reverse => {closure #a}
   list => {closure x}
```

The code of **list** has been shown before:

```
(define (list . x) x)
```

Here is an anonymous function that implements **list**:

```
(lambda x x)
```

When the argument list of a lambda function is a single variable, that varialbe binds a list containing all actual arguments.

So lambda functions can be variadic, just like named functions.

To implement a variadic function that expects some mandatory arguments, dotted argument lists are used:

```
         ((lambda (x . y) y)) => bottom
      ((lambda (x . y) y) 'a) => ()
   ((lambda (x . y) y) 'a 'b) => '#b
((lambda (x . y) y) 'a 'b 'c) => '#bc
```

Lambda functions are totally equivalent to named functions.

In fact, they are one and the same concept.

```
(define (pizza top) (list 'pizza 'with top))
```

is just a short form of writing

```
(define pizza (lambda (top) (list 'pizza 'with top)))
```

A named function is nothing but a variable bound to a lambda function.

2.5.1 bound and free variables

Here is a definition:

```
(define (f x) (cons x :f))
```

A variable that occurs in the argument list of a function is said to be *bound in that function.*

The variable *x* is bound in the function

```
(lambda (x) (cons x :f))
```

but the variable *cons* is not bound inside of that function.

A variable that does not occur in the argument list of a function but *does* occur in the term of the same function is said to be *free in that function.*

F is free in

```
(lambda (x) (f x))
```

A variable is said to be bound in a function because the variable gets bound to a value when the function is called.

But what values do free variables get?

It depends on the *lexical context* of the function.

The (lexical) context of

```
(lambda (x) (f x))
```

is the *global* context. The global context is the context in which **define** binds its values.

In the global context, *f* was bound to the function

```
(lambda (x) (cons x :f))
```

by the **define** at the beginning of this section. Therefore *f* is bound to the same function inside of

```
(lambda (x) (f x))
```

BTW: the variable *cons* in the function

```
(define (f x) (cons x :f))
```

is bound to the primitive **cons** function. The primitive functions of zenlisp are defined by the system itself when it starts. They are also defined in the global context.

2.6 local contexts

Here is an expression with a *local context*:

```
(let ((topping-1 'extra-cheese)
      (topping-2 'pepperoni))
  (list 'pizza 'with topping-1 'and topping-2))
```

The variables *topping-1* and *topping-2* are created locally inside of **let**.

Let is a pseudo function of two arguments.

The first argument is called its *environment* and the second one its *body* (or *term*).

The environment of **let** is a list of two-member lists:

$$
\begin{array}{l}
((name_1\ value_1) \\
\ \ \ldots \\
\ (name_n\ value_n))
\end{array}
$$

Let evaluates all values and then binds each value to the name associated with that value.

The term is evaluated inside of the context created by binding the values to names locally, and the normal form of the term is returned:

```
(let ((x 'heads) (y 'tails))
  (cons x y))
=> '(heads . tails)
```

Here is a **let** whose body is another **let**:

```
(let ((x 'outer))
  (let ((x 'inner)
        (y x))
    (list x y)))
=> '(inner outer)
```

There are two local contexts in this expression, an *inner* one and an *outer* one.

A context that is *created by* a **let** is called its *inner context*.

The context in which **let** occurs is called its *outer context*.

Let evaluates all values of its environment in its *outer* context:

```
(let ((x_outer 'outer))
  (let ((x_inner 'inner)
        (y x_outer))
    (list x y)))
=> '(inner outer)
```

In case of a duplicate symbol, the term of the inner **let** can only refer to the inner instance of that symbol, and the environment of the inner **let** can only refer to its outer instance.

BTW, **let** is nothing but an alternative syntax for the application of an anonymous function:

```
(let ((hd 'heads)    =   ((lambda (hd tl)
      (tl 'tails))          (cons hd tl))
  (cons hd tl))           'heads
                          'tails)
```

This equivalence explains neatly why values in environments are evaluated in outer contexts.

You would not expect a lambda function to evaluate its arguments in its inner context, would you?

2.6.1 closures

In nested local contexts, inner values shadow outer values:

```
(let ((v 'outer))
  (let ((v 'inner))
    v))
=> 'inner
```

But what happens when a local variable occurs as a free variable in a lambda function?

```
(let ((v 'outer))
  (let ((f (lambda () v)))
    (let ((v 'inner))
      (f))))
=> 'outer
```

What you observe here is a phenomenon called *lexical scoping*.

The value of the free variable *v* inside of the function *f* depends on the lexical context in which the function is *defined*.

The *definition* of *f* takes place in a context where *v* is bound to `'outer`, and the function *f* *memorizes* this binding.

The re-definition of *v* has no effect on it.

A function that memorizes the lexical values of its free variables is called a *closure*.

Here is a function that creates a closure:

```
(define (create-conser x)
  (lambda (y) (cons x y)))
```

The closure returned by the function conses *x* to its argument.

The value of *x* is specified when the closure is created:

```
(define cons-cherry (create-conser 'cherry))
(define cons-lemon (create-conser 'lemon))
```

The closures memorize the parameters passed to *create-conser*:

```
(cons-cherry 'pie) => '(cherry . pie)
(cons-lemon 'juice) => '(lemon . juice)
```

Alright, once again in slow motion:

```
((lambda (x) (lambda (y) (cons x y))) 'lemon)
—> (lambda (y) (cons 'lemon y))
```

This is a step called *beta reduction.* It is not a reduction in the sense of zenlisp, but a more abstract concept, as indicated by the generic arrow above.

Beta reduction replaces each variable of a lambda function which is free in the term of that function with the corresponding actual argument:

```
((lambda (y) (cons 'lemon y)) 'juice) -> (cons 'lemon 'juice)
```

Y is free in **`(cons 'lemon y)`**, so it is replaced by the value associated with *x*.

Things are different here:

```
((lambda (x) (lambda (x) x)) 'foo) —> (lambda (x) x)
```

Because *x* is bound in **`(lambda (x) x)`**, it is not replaced with `'foo`.

This is a mixed scenario:

```
((lambda (x) (list x (lambda (x) x))) 'foo) —> (list 'foo (lambda (x) x))
```

The first *x* is free and therefore replaced; the second one is bound in a function and hence left alone.

Beta reduction is a transformation of the *lambda calculus* (LC). LC is the mathematical foundation of all LISPy languages.

Beta reduction is a formal model for closures and parameter binding in zenlisp function calls.

2.6.2 recursive functions

Here is a simple function:

```
(define (d x)
  (or (atom x) (d (cdr x))))
```

Whatever you pass to *d*, it always returns **`:t`**:

```
(d 'x) => :t
(d '#xyz)
-> (d '#yz)
-> (d '#z)
-> (d ())
=> :t
```

Here is a not so simple question:

If **`lambda`** closes over all of its free variables and

```
(define (d x) (or (atom x) (d (cdr x))))
```

is equivalent to

```
(define d (lambda (x) (or (atom x) (d (cdr x)))))
```

then **`lambda`** must close over *d* before *d* gets bound to the resulting closure.

Then how can *d* recurse?
Obviously, it can.

The answer is that *d* is not really a closure at all (although it looks like one).

Whenever **`define`** binds a name to a lambda function, it stops **`lambda`** from capturing its free variables.

The result is a closure with an *empty environment*.

Because the function that is bound to *d* does not have a local value for *d*, it resorts to the global binding of *d*.

This method is called *dynamic scoping*.

It is called ‘‘dynamic’’, because it allows to change the values of free variables dynamically.

Here is dynamic scoping in action:

```
(define food 'marmelade)
(define (get-food) food)
(get-food) => 'marmelade
(define food 'piece-of-cake)
(get-food) => 'piece-of-cake
```

Because dynamic scoping allows to bind values to symbols *after* using the symbols in functions, it can even be used to create *mutually recursive* functions:

```
(define (d1 x) (or (atom x) (d2 (cdr x))))
(define (d2 x) (or (atom x) (d1 (cdr x))))
(d1 '#xyz) => :t
```

Two functions f and g are mutually recursive if f calls g and g calls f.

BTW, you can use **`let`** with an empty environment to make **`define`** create a closure:

```
(define food 'marmelade)
(define get-food (let () (lambda () food)))
(get-food) => 'marmelade
(define food 'piece-of-cake)
(get-food) => 'marmelade
```

However, this method renders recursive functions impossible:

```
(define dc (let () (lambda (x)
                     (or (atom x)
                         (dc (cdr x))))))
(dc '#xyz) => bottom
```

2.6.3 recursive closures

`Let` can bind trivial functions:

```
(let ((cons-lemon
        (lambda (x)
          (cons 'lemon x))))
  (cons-lemon 'cake))
=> '(lemon . cake)
```

`Let` can *not* bind recursive functions, because the function closes over its own name *before* it is bound to that name:

```
(let ((d (lambda (x)
           (or (atom x)
               (d (cdr x))))))
  (d '#xyz))
=> bottom
```

`Letrec` can bind *recursive functions*:

```
(letrec ((d (lambda (x)
          (or (atom x)
              (d (cdr x))))))
  (d '#xyz))
=> :t
```

This is why it is called `let`*`rec`*.

Using **`letrec`**, functions like **`reverse`** [page 21] can be packaged in one single **`define`**:

```
(define (reverse a)
  (letrec
    ((reverse2
       (lambda (a r)
         (cond ((null a) r)
               (t (reverse2 (cdr a)
                            (cons (car a) r)))))))
    (reverse2 a ())))
```

Even *mutual recursion* can be implemented using **`letrec`**:

```
(letrec ((d1 (lambda (x)
               (or (atom x) (d2 (cdr x)))))
         (d2 (lambda (x)
               (or (atom x) (d1 (cdr x))))))
  (d1 '#xyz))
=> :t
```

The only difference between **`let`** and **`letrec`** is that **`letrec`** fixes recursive references in its environment after binding values to symbols.

Therefore

Local functions should be bound by `letrec`, local data by `let`.

2.6.4 recursive bindings

Here is an expression reducing to a closure:

```
(lambda () f)
```

Applications of the closure reduce to no value, because *f* has no value:

```
((lambda () f)) => bottom  ; make sure that F is unbound!
```

Zenlisp has a set of *meta functions* that modify the state of the interpreter. The meta function application

```
(closure-form env)
```

makes the interpreter display the lexical environments of closures when printing their normal forms:

```
(lambda () f) => (closure () f ((f . {void})))
```

The closure prints in round brackets because its full representation is unambiguous.

All four parts of the closure print now: the **closure** keyword, the argument list, the body, and the lexical environment captured by **lambda**.

The environment is stored in an *association list* (a.k.a. *alist*).

An association list is a list of pairs, where each car part of a pair is a key and each cdr part is a value *associated* with that key:

((*key*$_1$. *value*$_1$) ... (*key*$_n$. *value*$_n$))

In lexical environments, variable names are keys and the values associated with the keys are the values of those variables.

The **assq** and **assoc** functions are used to retrieve associations from alists:

```
(define alist '((food  . orange) (drink . milk)))
(assq 'drink alist) => '(drink . milk)
 (assq 'soap alist) => :f
```

Assoc is related to **assq** in the same way as **member** is related to **memq** [page 15]:

```
 (assq '(key) '(((key) . value))) => :f
(assoc '(key) '(((key) . value))) => '((key) . value)
```

The problem with the alist of the closure

```
(closure () f ((f . {void})))
```

is that *f* is associated with no specific value -- even if the closure itself is associated with the symbol *f* (the form **{void}** indicates that *f* is not bound at all):

```
(let ((f (lambda () f))) f)
=> (closure () f ((f . {void})))
```

We mean to refer to the inner *f*, but **lambda** closes over the outer *f*.

Letrec uses a function called **recursive-bind** to fix environments containing recursive references:

```
(recursive-bind '((f . (closure () f ((f . wrong))))))
```

It does not matter what *f* is bound to in the inner context, because that is exactly what **recursive-bind** will change.

In the resulting structure, `wrong` is replaced with a reference to the value of the outer *f*:

```
(recursive-bind '(($f_{outer}$ . (closure () $f_{inner}$ (($f_{inner}$ . wrong))))))
               => '(($f_{outer}$ . (closure () $f_{inner}$ (($f_{inner}$ . $f_{outer}$)))))
```

Because *f* now contains a reference to *f*, it is a cyclic structure.

When you reduce above application of **recursive-bind** with **closure-form** set to `env`, the interpreter will attempt to print an infinite structure:

```
(recursive-bind '((f . (closure () f ((f . wrong))))))
=> '((f . (closure () f
            ((f . (closure () f
                    ((f . (closure () f
                            ((f . ...
```

This is why only the argument lists of closures print by default. You can restore this behaviour by applying

```
(closure-form args)
```

2.7 higher-order functions

Here is a *higher-order function*:

```
(define (compose f g)
  (lambda x (f (apply g x))))
```

A higher-order function is a function that takes a function as an argument and/or reduces to a function.

The *create-conser* function [page 37] introduced earlier also was a higher-order function.

The *compose* function takes two functions *f* and *g* as arguments and creates an anonymous function that implements the composition of *f* and *g*.

The **cadr** function, which extracts the second element of a list, can be implemented using *compose*:

```
(compose car cdr) => {closure #x}
((compose car cdr) '(pizza with pepperoni)) => 'with
```

Of course, you would probably write **cadr** this way:

```
(define (cadr x) (car (cdr x)))
```

So here comes a more interesting example.

Filter extracts members with a given property from a flat list. The property is tested by applying the predicate *p* to each member: [3]

[3] Writing a tail-recursive version of *filter* is left as an exercise to the reader.

```
(define (filter p lst)
  (cond ((null lst) ())
        ((p (car lst))
          (cons (car lst)
                (filter p (cdr lst))))
        (t (filter p (cdr lst)))))
```

Filter is a higher-order function because it expects a function — a predicate to be precise — in the place of *p*. It extracts members of a list that satisfy that predicate:

```
(filter atom '(orange '#xyz juice (a . b))) => '(orange juice)
```

A negative filter that removes all elements with a given property can be implemented on top of *filter* using a closure:

```
(define (remove p lst)
  (filter (lambda (x) (not (p x)))
          lst))
(remove atom '(orange #xyz juice (a . b))) => '(#xyz (a . b))
```

`Not` implements the logical "not":

```
(define (not x) (eq x :f))
```

The function

```
(lambda (x) (not (p x)))
```

is in fact a useful higher-order function on its own. Here is a slightly improved version:

```
(define (complement p)
  (lambda x (not (apply p x))))
```

It turns a predicate into its complement:

```
                      (atom 'x) => :t
          ((complement atom) 'x) => :f
             (eq 'apple 'orange) => :f
((complement eq) 'apple 'orange) => :t
```

2.7.1 mapping

`Map` maps functions over lists:

```
(map (lambda (x) (cons 'lemon x))
     '(cake juice fruit))
=> '((lemon . cake) (lemon . juice) (lemon . fruit))
```

`Map` implements list manipulation and flow control at the same time.

Here is a function that uses **`map`** to compute the depth of a list: [4]

```
(require '~nmath) ; load natural math package
(define (depth x)
  (cond ((atom x) '#0)
        (t (+ '#1 (apply max (map depth x))))))
```

`Map` can map functions over any number of lists:

```
(map car '(#ab #cd #ef)) => '#ace

(map cons '#ace '#bdf) => '((a . b) (c . d) (e . f))

(map (lambda (x y z) (append x '#- y '#- z))
     '(#lemon     #cherry)
     '(#chocolate #banana)
     '(#ice-cream #shake))
=> '(#lemon-chocolate-ice-cream
     #cherry-banana-shake)
```

Map-car is a simplified version of **`map`** that accepts only unary functions (functions of one variable) and only one single list:

```
(define (map-car f a)
  (letrec
    ((map-car2
       (lambda (a r)
         (cond ((null a) (reverse r))
               (t (map-car2 (cdr a)
                            (cons (f (car a)) r)))))))
    (map-car2 a ())))
```

The *any-null* predicate checks whether any of the data contained in a list is equal to **`()`**:

```
(define (any-null lst)
  (apply or (map-car null lst)))
```

Mapping **`null`** over a list gives a list of truth values:

```
(map-car null '(#x #y () #z)) => '(:f :f :t :f)
```

Applying **`or`** to the resulting list gives truth if at least one member of the list is not **`:f`**.

The *car-of* and *cdr-of* functions map a list of lists to a list of car and cdr parts respectively:

```
(define (car-of x) (map-car car x))
(define (cdr-of x) (map-car cdr x))
```

Any-null, *car-of*, and *cdr-of* are used to implement **`map`**:

```
(define (map f . a)
  (letrec
```

4 **+** computes the sum of a list, **`max`** extracts the maximum of a list. `'#0` is zenlisp's way to express the number 0.

```
    ((map2
       (lambda (a b)
         (cond ((any-null a) (reverse b))
               (t (map2 (cdr-of a)
                        (cons (apply f (car-of a)) b)))))))
    (map2 a ()))))
```

Because *any-null* checks whether *any* of the sublists of its argument is **()**, **map** returns as soon as the end of the shortest list passed to it is reached:

```
(map cons '#ab '#abcd) => '((a . a) (b . b))
```

2.7.2 folding

Fold folds lists by combining adjacent members:

```
(fold cons 'a '(b c d)) => '(((a . b) . c) . d)
```

The first argument of **fold** is the function used to combine elements, the second argument is the "base" element, and the third one is the list of elements to combine. The base element is combined with the first member of the list.

When the list of members is empty, **fold** returns the base element:

```
(fold cons 'empty ()) => 'empty
```

Fold is used to iterate over lists.

Predicate-iterator is a higher-order function that uses **fold** to make a two-argument predicate variadic [5] (**Neq** is **(compose not eq)**):

```
(define (predicate-iterator pred)
  (let ((:fail ':fail))
    (let ((compare
           (lambda (a b)
             (cond ((eq a :fail) :fail)
                   ((pred a b) b)
                   (t :fail)))))
      (lambda (first . rest)
        (neq (fold compare first rest) :fail)))))
```

Here is **predicate-iterator** in action:

```
(define eq* (predicate-iterator eq))
(eq*) => bottom
(eq* 'lemon) => :t
(eq* 'lemon 'lemon) => :t
(eq* 'lemon 'lemon 'lemon) => :t
(eq* 'lemon 'lemon 'orange 'lemon) => :f
```

5 The **predicate-iterator** function has a minor flaw. See page 55 for details.

Should an application of *eq** to less than two arguments really reduce to **`:t`**? If not, how would you fix **`predicate-iterator`**?

`Fold` reduces a list by combining its members *left-associatively*:

```
(fold f 'a '(b c d))  =  (f (f (f a b) c) d)
```

An operation is left-associative, if multiple applications of the same operator *associate to the left*, i.e. bind stronger to the left.

The mathematical ‘‘minus’’ operator is left-associative, because

$$a - b - c - d = (((a - b) - c) - d)$$

To fold the members of a list right-associatively, the **`fold-r`** function is used:

```
(fold-r f 'a '(b c d))  =  (f a (f b (f c d)))
```

An example for right-associative operators is mathematical exponentiation:

$$a^{b^{c^d}} = a^{(b^{(c^d)})}$$

The following reductions illustrate the difference between **`fold`** and **`fold-r`**.

```
  (fold cons 'a '(b c d)) => '(((a . b) . c) . d)
(fold-r cons 'a '(b c d)) => '(b . (c . (d . a)))
                           = '(b c d . a)
```

3. rather esoteric aspects

3.1 numeric functions

Before a zenlisp program can work with numbers, it has to *require* one of the math packages:

```
(require '~nmath)
```

`Nmath` is a package containing functions for computations with *natural numbers* (plus zero).

Zenlisp uses lists of digits to represent natural numbers. Here are some examples:

```
'#0  '#3  '#1415  '#926535
```

Here is the **`length`** function:

```
(define (length x)
  (letrec
    ((length2
       (lambda (x r)
         (cond ((null x) r)
               (t (length2 (cdr x) (+ '#1 r)))))))
    (length2 x '#0)))
```

`Length` computes the length of a list:

```
             (length ()) => '#0
(length '(orange juice)) => '#2
   (length '#4142135623) => '#10
```

It uses the math function **+**, which adds numbers:

```
(+) => '#0
(+ '#5) => '#5
(+ '#5 '#7) => '#12
(+ '#5 '#7 '#9) => '#21
```

Like many other math functions, **+** is *variadic*.

Passing zero arguments to **+** yields `'#0`, the neutral element of the mathematical "plus" operation.

Here are some other math functions:

```
(*) => '#1
(* '#5 '#7 '#9) => '#315
(- '#7 '#3 '#4) => '#0
```

The **`*`** function computes the product of its arguments and **–** computes their difference.

Integer division is performed by the **divide** function:

```
(divide '#17 '#3) => '(#5 #2)
```

It returns a list containing the integer quotient and the division remainder of its arguments:

```
(divide dividend divisor) => '(quotient remainder)
```

When only the quotient or the remainder of two numbers is of interest, the **quotient** and **remainder** functions are useful:

```
 (quotient '#17 '#3) => '#5
(remainder '#17 '#3) => '#2
```

The division remainder of two numbers *a* and *b* is defined as
(- *a* **(*** **(quotient** *a b***)** *b***))**.

(Expt x y) computes *x* to the power of *y*:

```
(expt '#3 '#100) => '#515377520732011331036461129765621272702107522001
```

Because zenlisp implements numbers as lists of digits, precision is only limited by time and memory.

This approach is known as *bignum arithmetics* (big number arithmetics). It guarantees that numeric expressions always yield a mathematically correct result (which includes, of course, the possibility of returning *bottom*).

3.1.1 numeric predicates

Here is the code of the *ngcd* function:

```
(define (ngcd a b)
  (cond ((zero b) a)
        ((zero a) b)
        ((< a b) (ngcd a (remainder b a)))
        (t (ngcd b (remainder a b)))))
```

Ngcd computes the *greatest common divisor* of two natural numbers:

```
  (ngcd '#12 '#6) => '#6
(ngcd '#289 '#34) => '#17
 (ngcd '#17 '#23) => '#1
```

Of course you would use the **gcd** function of one of the zenlisp math packages to compute the gcd, but *ngcd* uses two interesting functions.

The **zero** function tests whether its argument is equal to the number 0:

```
(zero '#0) => :t
(zero '#1) => :f
```

The argument of **`zero`** must be a *number*:

```
(zero 'non-number) => bottom
```

`Zero` is a *numeric predicate.*

It can be expressed (albeit less efficiently) using another numeric predicate:

```
(define (zero x) (= x '#0))
```

The **=** predicate tests whether its arguments are numerically equal:

```
(= '#3 '#3 '#3) => :t
  (= '#23 '#17) => :f
```

Here are some other numeric predicates:

```
 (< '#1 '#2 '#3) => :t
 (> '#3 '#2 '#1) => :t
(<= '#1 '#2 '#2) => :t
(>= '#2 '#1 '#1) => :t
```

They test for the properties "*less than*", "*greater than*", "*less than or equal to*", and "*greater than or equal to*", respectively.

The following relation holds for each comparative numeric predicate *R*:

$(R\ a_1\ \ldots\ a_n) \quad = \quad (\texttt{and}\ (R\ a_1\ a_2)\ \ldots\ (R\ a_{n-1}\ a_n))$

So, for example,

```
(< a b c d)
```

can be written as

```
(and (< a b) (< b c) (< c d))
```

This is exactly the reason why there is *no* negative equivalence operator.

If there was such an operator (let us call it *not*=), the expression

$(not\texttt{=}\ a_1\ \ldots\ a_n)$

would translate to

$(\texttt{and}\ (not\texttt{=}\ a_1\ a_2)\ \ldots\ (not\texttt{=}\ a_{n-1}\ a_n))$

while

$(\texttt{not}\ (\texttt{and}\ (=\ a_1\ an_2)\ \ldots\ (=\ a_{n-1}\ a_n)))$

equals

$(\texttt{or}\ (not\texttt{=}\ a_1\ a_2)\ \ldots\ (not\texttt{=}\ a_{n-1}\ a_n))$

3.1.2 integer functions

When using the **nmath** package, negative results cannot occur:

```
(- '#0 '#1) => bottom
```

Loading the **imath** (integer math) package extends the domain and/or range of the natural math functions to include negative numbers:

```
(require '~imath)
(- '#0 '#1) => '#-1
```

Loading **imath** loads **nmath** automatically.

After loading the integer math package, most functions discussed in the previous section accept negative arguments, too:

```
        (+ '#5 '#-7) => '#-2
       (* '#-5 '#-5) => '#25
(quotient '#17 '#-3) => '#-5
```

The **-** function is extended to handle single arguments, implementing the “negate” operator:

```
(- '#27182) => '#-27182
```

In addition to the **remainder** function **imath** introduces the **modulo** function. The modulus of *a* and *b* is defined as: [6]

$a - floor(a/b) \times b$

The results of **remainder** and **modulo** differ only when their arguments *a* and *b* have different signs:

argument a	argument b	remainder	modulus
'#+23	'#+5	'#+3	'#+3
'#+23	'#-5	'#+3	'#-2
'#-23	'#+5	'#-3	'#+2
'#-23	'#-5	'#-3	'#-3

Because zenlisp lacks a “floor” function, **modulo** exploits the fact that

```
(modulo x y)   =   (+ y (remainder x y))
```

if *x* and *y* have different signs.

Here is **modulo**:

```
(define (modulo a b)
  (let ((rem (remainder a b)))
```

6 The “floor” of *x* denotes the largest integer that is not larger than *x*.

```
    (cond ((zero rem) '#0)
          ((eq (negative a) (negative b))
            rem)
          (t (+ b rem)))))
```

The **negative** predicate tests whether its argument (which must be numeric) is negative.

Eq is used as a *logical equivalence* operator in **modulo**:

```
(eq (negative a) (negative b))
```

reduces to truth if either both *a* and *b* are positive or both are negative.

3.1.3 rational functions

When using the **imath** package, results *between* zero and one cannot occur:

```
(expt '#2 '#-5) => bottom
```

Loading the **rmath** (rational math) package extends the domain and/or range of the integer math functions to include rational numbers:

```
(require '~rmath)
(expt '#2 '#-5) => '#1/32
```

The format of rational numbers is

'#*numerator*/*denominator*

where *numerator* and *denominator* are integer numbers.

Negative rational numbers have a negative numerator:

```
(- '#1 '#3/2) => '#-1/2
```

The **numerator** and **denominator** functions extract the individual parts of a rational number:

```
  (numerator '#-5/7) => '#-5
(denominator '#-5/7) => '#7
```

Even if the **rmath** package was loaded, the **divide** and **quotient** functions still perform integer division. The **/** function performs the division of rational numbers:

```
   (/ '#10 '#2) => '#5
   (/ '#20 '#6) => '#10/3
(/ '#1/2 '#1/2) => '#1
```

The **sqrt** function of the **nmath** package delivers the integer part of the square root of a number. **Rmath** extends this function to deliver a rational result:

```
(sqrt '#144) => '#12
  (sqrt '#2) => '#665857/470832
```

The precision of the rational **`sqrt`** function is controlled by the **`*epsilon*`** variable:

```
*epsilon* => '#10
```

An **`*epsilon*`** value of 5, for example, denotes a precision of 10^{-5}, i.e. when converted to fixed point notation, the result of **`sqrt`** has a precision of (at least) 5 decimal places. The default precision is 10^{-10}.

3.1.4 type checking and conversion

The **`number-p`** predicate checks whether a datum represents a number:

```
(number-p '#314) => :t
(number-p '#-159) => :t
(number-p '#-265/358) => :t
```

Any type of datum can be passed to **`number-p`**:

```
      (number-p 'marmelade) => :f
(number-p '(heads . tails)) => :f
```

The **`natural-p`**, **`integer-p`**, and **`rational-p`** predicates test whether a number has a specific type:

```
   (natural-p '#1) => :t
  (integer-p '#-1) => :t
(rational-p '#1/2) => :t
```

A datum passed to one of these predicates *must be a valid number*:

```
(integer-p 'sushi) => bottom
```

Also note that these predicates only check the syntax of the data passed to them. Therefore

```
 (natural-p '#+1) => :f
(integer-p '#1/1) => :f
 (rational-p '#5) => :f
```

Although `'#1/1` is an integer value, **`integer-p`** does not recognize it as such because it has the syntax of a rational number.

To check whether the *value* of a number has a specific type, the datum representing the number must be *normalized* first. This can be done by applying a neutral operation:

```
 (natural-p (+ '#0 '#+1)) => :t
(integer-p (* '#1 '#1/1)) => :t
```

However, this works only if the argument has the syntax of a ‘‘more complex’’ type than the one checked against:

```
(rational-p '(+ '#0 '#5)) => :f
```

Although `'#5` is a valid rational number, **`rational-p`** does not return truth because the normalized form of `'#5` is still `'#5` and not `'#5/1`.

This is what normalization does:

– reduce rationals to least terms;
– move the signs of rationals to the numerator;
– remove denominators of 1;
– remove plus signs from integers;
– remove leading zeroes.

Normalization reduces each number to its *least applicable type*.

Rationals are a superset of integers and integers are a superset of naturals.

The least type of a number is the type representing the smallest set that includes that number.

For instance, `'#-5/-5` is a rational number, but normalizing it yields a natural number:

```
(+ '#0 '#-5/-5)
-> '#-5/-5        ; added '#0
-> '#1/1          ; reduced to least terms
=> '#1            ; removed denominator of '#1
```

The **`natural`**, **`integer`**, and **`rational`** functions attempt to convert a number to a different type:

```
 (natural '#+5) => '#5
  (integer '#5) => '#5
(rational '#-5) => '#-5/1
```

In case the requested conversion cannot be done, these functions reduce to bottom, thereby implementing numeric *type checking*:

```
 (natural '#-1) => bottom
(integer '#1/2) => bottom
```

The **`rmath`** package is required for the following type conversion to work:

```
(require '~rmath)
(integer '#4/2) => '#2
```

It will not work when using just **`imath`**:

```
(require '~imath)
(integer '#4/2) => bottom
```

Type conversion functions are used to guard functions against arguments that are not in their domains.

Here is a higher-order function that turns a two-argument math function into a variadic math function:

```
(define (arithmetic-iterator conv fn neutral)
  (lambda x
    (cond ((null x) neutral)
          (t (fold (lambda (a b)
                     (fn (conv a) (conv b)))
                   (car x)
                   (cdr x))))))
```

The *fn* argument is the function to convert, *conv* is one of the type conversion functions, and *neutral* is a base value for **`fold`**.

Let *n+* be an unguarded binary function (a function of two variables) for adding natural numbers:

```
    (n+ '#1 '#2) => '#3
  (n+ '#+1 '#+2) => bottom
(n+ '#1 '#2 '#3) => bottom
```

`Arithmetic-iterator` turns this function into a guarded variadic function:

```
((arithmetic-iterator natural n+ '#0) '#+1 '#+2 '#+3) => '#6
     ((arithmetic-iterator natural n+ '#0) '#+0 '#-1) => bottom
```

`Arithmetic-iterator` is actually used to implement most of the zenlisp math functions.

3.2 side effects

The *effect* of a *function* is to map values to values:

```
(null ()) => :t
 (null x) => :f  ; for any x that does not equal ()
```

The effect of **`null`** is to map **`()`** to **`:t`** and any other value to **`:f`**.

Each time a function is applied to the same actual argument(s) it returns the same result:

```
(atom 'x) => :t
(atom 'x) => :t
(atom 'x) => :t
```

This is a fundamental property of a function.

An effect that cannot be explained by mapping values to values is called a *side effect*.

`Define` is known to have a side effect:

```
(define marmelade 'medium-cut-orange) => 'marmelade
marmelade => 'medium-cut-orange
(define marmelade 'fine-cut-orange) => 'marmelade
marmelade => 'fine-cut-orange
```

The mutation of the value of the variable *marmelade* cannot be explained by mapping `marmelade` and `'fine-cut-orange` to `'marmelade`.

`Define` does not even deliver the same value for each pair of arguments: [7]

```
(define define :f) => 'define
(define define :f) => bottom
```

Because the first application of **`define`** changes the value of *define* itself to **`:f`**, the second form is not even a valid expression.

A function that has a side effect is a *pseudo function*, but not every pseudo function has a side effect. **`Cond`**, **`quote`**, **`let`**, and **`lambda`** are examples for pseudo functions without side effects.

3.2.1 subtle side effects

Neither **`cons`** nor **`eq`** has a side effect, but by bending the rules a little bit, they can be used to construct one.

`Cons` is guaranteed to deliver a *fresh* pair.

`Eq` compares data by comparing their locations in the memory of the computer. This is, of course, an implementation detail. You do not need to memorize it, but its implications are interesting:

`(eq (cons x y) (cons x y)) => :f` for any values of *x* and *y*.

This is the exception to the rule that passing two pairs to **`eq`** yields bottom:

Two applications of `cons` can never yield identical results.

Hence **`cons`** can create an absolutely *unique instance* of a datum:

```
(define unique-instance (cons 'unique-instance ()))
```

and **`eq`** can identify that instance:

```
(eq unique-instance unique-instance) => :t
(eq unique-instance '(unique-instance)) => :f
 unique-instance => '(unique-instance)
```

This effect cannot be explained by mapping values to values, so it obviously is a side effect.

The creation of unique instances is the only reason why this side effect is described here. It can be used to overcome a subtle limitation of the **`predicate-iterator`** function [page 45].

`Predicate-iterator` uses the symbol `:fail` to represent failure. Hence it can deliver a wrong value when the function returned by it is applied to **`'`**`:fail` itself:

```
((predicate-iterator equal) ':fail ':fail) => :f
```

7 If you actually try this, you will have to restart zenlisp to restore the original behaviour.

No matter what symbol you would invent in the place of `:fail`, the function created by **`predicate-iterator`** would never handle that symbol correctly.

What you need is a *unique instance* of the datum representing failure:

```
(define (predicate-iterator pred)
  (let ((:fail (cons ':fail ())))
    (let ((compare
            (lambda (a b)
              (cond ((eq a :fail) :fail)
                    ((pred a b) b)
                    (t :fail)))))
      (lambda (first . rest)
        (neq (fold compare first rest) :fail)))))
```

In the improved version of **`predicate-iterator`**, *:fail* binds to a unique instance of `'(:fail)`. Therefore, the function can even handle other instances of that datum correctly:

```
((predicate-iterator equal) '(:fail) '(:fail)) => :t
```

3.2.2 evaluation

The **`eval`** function interprets zenlisp expressions:

```
(eval '(letrec
         ((n (lambda (x)
               (cond ((atom x) ())
                     (t (cons 'i (n (cdr x))))))))
         (n '(1 2 3 4 5))))
=> '#iiiii
```

Note that the argument of **`eval`** is a *list* and therefore it is not evaluated before it is passed to **`eval`**. It is the **`eval`** function that gives meaning to a form, thereby turning it into an expression.

`Eval` reduces to bottom, if the datum passed to it does not constitute a valid expression:

```
(eval '(cons 'x)) => bottom
```

`Eval` has a side effect, if the expression interpreted by it has a side effect:

```
(eval '(define bar 'foo)) => 'bar
bar => 'foo
```

3.3 metaprogramming

Here is a trivial *metaprogram*:

```
(list 'lambda '(x) 'x) => '(lambda (x) x)
```

A metaprogram is a program that writes or rewrites a program.

Programs are expressions, and expressions are a subset of forms.

Every program can be turned into a datum by *quoting* it:

```
(quote (lambda (x) x)) => '(lambda (x) x)
```

Here is another simple meta program:

```
((lambda #x (list x (list 'quote x)))
'(lambda #x (list x (list 'quote x))))
```

What is interesting about this kind of program is that its normal form is equal to its own code; *it reduces to itself*: [8]

```
((lambda #x (list x (list 'quote x)))
'(lambda #x (list x (list 'quote x))))
=> ((lambda #x (list x (list 'quote x)))
   '(lambda #x (list x (list 'quote x))))
```

Such programs are widely known as *quines*. They are named after the philosopher W.V.O. Quine, who did a lot of research in the area of indirect self reference.

The above program inserts the quoted "half" of its code in the place of *x* in the form

```
(list x (list 'quote x))
```

thereby creating code that applies this half to a quoted copy of itself. The self reference is indirect because the expression does not reference itself, but *creates* a reference to itself.

Here is another classic self-referential expression:

```
((lambda #x #xx) (lambda #x #xx))
```

Can you deduce its normal form? Does it have one?

3.3.1 programs hacking programs

Here is a not so trivial metaprogram:

```
(define (let->lambda let-expr)
  (let ((env (cadr let-expr)))
    (let ((vars (map car env))
          (args (map (lambda (x) (unlet (cadr x)))
                     env))
          (body (unlet (caddr let-expr))))
      (append (list (list 'lambda vars body))
              args))))
```

8 Argument lists are condensed in this example, because this is how the zenlisp printer prints them. This has no effect on the semantics, though. For example, **`(lambda (x) (x x))`** is equal to **`(lambda #x #xx)`**.

```
(define (unlet x)
  (cond ((atom x) x)
        ((eq (car x) 'quote) x)
        ((eq (car x) 'let) (let->lambda x))
        ((eq (car x) 'lambda)
          (list 'lambda
                (cadr x)
                (unlet (caddr x))))
        (t (map unlet x))))
```

Let->lambda converts (a datum representing) a **let** expression to (a datum representing) the application of a lambda function:

```
(let->lambda '(let ((x 'heads)
                    (y 'tails))
                (cons x y)))
=> '((lambda #xy
       (cons x y))
     'heads 'tails)
```

The *unlet* function traverses (a datum representing) a zenlisp expression and replaces each **let** contained in it:

```
(unlet '(let ((y (let ((a '(orange)))
                   a))
              (z '(cookies)))
          (let ((x '(i like)))
            (append x y z))))
=> '((lambda #yz
       ((lambda #x
          (append x y z))
        '(i like)))
     ((lambda #a a)
      '(orange))
     '(cookies))
```

If nothing else, this program demonstrates that **let** is in some situations much more readable than **lambda**.

Unlet recurses through **map**. Recursion through map is always a *structural recursion.*

BTW, the default case **(map unlet x)** *cannot be replaced with*

```
(cons (unlet (car x))
      (unlet (cdr x)))
```

because doing so would break the test for quotation. Given

x = '(list quote x)

the above solution would lead to

```
(cons (unlet 'list)
      (unlet '(quote x)))
```

which in turn would result in *unlet* treating *x* as a quoted symbol, which it is not.

3.3.2 beta reduction by substitution

This section introduces a slightly more elaborate metaprogram.

Lambda-rename renames the variables of lambda expressions in such a way that each variable gets a unique name:

```
(lambda-rename '(lambda (x) (lambda (y) (cons x y))))
=> '(lambda (x:0) (lambda (y:1) (cons x:0 y:1)))
```

Its purpose is to resolve conflicts between variables:

```
(lambda-rename '(lambda (x) (list x (lambda (x) x))))
=> '(lambda (x:0) (list x:0 (lambda (x:1) x:1)))
```

Lambda-rename uses a few helper functions.

The *add* function adds a colon and a number to a symbol name:

```
(require '~nmath) ; this will be needed later
(define (add name level)
  (implode (append (explode name)
                   '#:
                   level)))
```

It uses the **`implode`** function to create a *new symbol name* from a list of single-character names:

```
(implode '(n e w - n a m e)) => 'new-name
```

Zenlisp numbers already *are* lists of symbols:

```
(cons 'digits: '#31415) => '(digits: 3 1 4 1 5)
```

In order to append `'#:` and a number to an existing symbol, the existing symbol must be exploded first.

The **`explode`** function implements the reverse operation of **`implode`**: [9]

```
(explode 'symbol) => '(s y m b o l)
```

Lambda-rename keeps substitutions of old and new names in an association list, e.g:

```
'((y . y:1) (x . x:0))
```

Ext-sub takes a list of variable names, an alist, and a number. It adds a new substitution for each

9 Zenlisp will print `'(s y m b o l)` as `'#symbol`.

variable name passed to it:

```
(ext-sub '((x . x:2)) '(a b c) '#3)
=> '((c . c:3) (b . b:3) (a . a:3) (x . x:2))
```

Here is *ext-sub*:

```
(define (ext-sub sub vars level)
  (letrec
    ((add-var
       (lambda (name alist)
         (cons (cons name (add name level))
               alist))))
    (cond ((null vars) sub)
          ((atom vars) (add-var vars sub))
          (t (ext-sub (add-var (car vars) sub)
                      (cdr vars)
                      level)))))
```

Ext-sub handles variadic argument lists properly:

```
(ext-sub '() '(x . y) '#1) => '((y . y:1) (x . x:1))
```

The *subst* function is similar to **assoc**:

```
(define (subst name sub)
  (let ((v (assq name sub)))
    (cond (v (cdr v))
          (t name))))
```

Its only difference to **assoc** is that it returns the value in case of success and the key if no matching association is found:

```
   (subst 'x '((x . x:0))) => 'x:0
(subst 'cons '((x . x:0))) => 'cons
```

Rename-vars traverses a datum representing an expression and renames all variables of lambda functions in both their argument lists and bodies:

```
(define (rename-vars expr sub level)
  (cond ((atom expr) (subst expr sub))
        ((eq (car expr) 'quote) expr)
        ((eq (car expr) 'lambda)
          (let ((vars (cadr expr))
                (body (caddr expr)))
            (let ((new-sub (ext-sub sub vars level)))
              (list 'lambda
                    (rename-vars vars new-sub level)
                    (rename-vars body
                                 new-sub
                                 (+ '#1 level))))))
        (t (map-car-i (lambda (x)
                        (rename-vars x sub level))
                      expr))))
```

Because *rename-vars* has to be able to process variadic argument lists of **`lambda`**, it cannot recurse through **`map`**. It uses a special version of *map-car* [page 44] that is able to process dotted (improper) lists (hence the trailing "i" in its name):

```
(define (map-car-i f a)
  (letrec
    ((map-car-i2
       (lambda (a r)
         (cond ((null a) (reverse r))
               ((atom a) (append (reverse r) (f a)))
               (t (map-car-i2 (cdr a)
                              (cons (f (car a)) r)))))))
    (map-car-i2 a ())))
```

Two of the arguments of *rename-vars* are constant, so here is a wrapper that provides them:

```
(define (lambda-rename expr)
  (rename-vars expr () '#0))
```

After renaming the variables of lambda expressions using *lambda-rename*, *beta reduction* is nothing more than a simple substitution.

The *subst-vars* function substitutes each symbol of a given expression with the value associated with that symbol:

```
(define (subst-vars expr sub)
  (cond ((atom expr) (subst expr sub))
        ((eq (car expr) 'quote) expr)
        (t (map-car-i (lambda (x)
                        (subst-vars x sub))
                      expr))))
```

Using *lambda-rename* and *subst-vars*, beta reduction can be done without using closures:

```
(define (beta-reduce app)
  (let ((app (lambda-rename app)))
    (let ((vars (cadar app))
          (args (cdr app))
          (body (caddar app)))
      (subst-vars body (map cons vars args)))))
```

Instead of creating closures, *beta-reduce substitutes* free variables with their values:

```
(beta-reduce '((lambda (x) x) v))
=> 'v
(beta-reduce '((lambda (f) (lambda (x) (f x))) not))
=> '(lambda (x:1) (not x:1))
(beta-reduce '((lambda (x) (list x (lambda (x) x))) v))
=> '(list v (lambda (x:1) x:1))
```

You can use **`eval`** to interpret the output of metaprograms:

```
(eval (beta-reduce '((lambda (x) (list x (lambda (x) x))) 'v)))
=> '(v {closure (x:1) x:1})
```

3.4 packages

This is a zenlisp *package*:

```
(define square :t)           ; name the package
(require '~rmath)            ; declare dependencies
(define (square x) (* x x))  ; package body
```

The **require** function loads a package with the given name only if the name is not bound. For instance

```
(require '~rmath)
```

loads **rmath** only, if the symbol **rmath** is not bound to any value. If the symbol is bound, **require** assumes that the package already has been loaded.

The leading tilde ("~") makes zenlisp load a package from a standard location.

Because **require** loads packages only once, recursive references pose no problem. Loading a file named `foo.l` containing the package

```
(define foo :t)
(require 'foo)  ; require myself
```

will just load `foo`. No infinite recursion will occur.

part two algorithms

4. list functions

4.1 heads and tails

Just for warming up, how do you find out whether a list *x* is the head of a list *y*, e.g.:

```
(headp '#a '#abc) => :t
(headp '#ab '#abc) => :t
(headp '#abc '#abc) => :t
(headp '#abcd '#abc) => :f
```

A list *x* is the head of a list *y*, if their leading members up to the length of *x* are equal. Here is *headp*, which performs this test:

```
(define (headp x y)
  (cond ((null y) (null x))
        ((null x) :t)
        (t (and (equal (car x) (car y))
                (headp (cdr x) (cdr y))))))
```

According to *headp*, the empty list is the head of any list:

```
(headp () '(foo bar baz)) => :t
```

Do you think this is the right way to handle the empty ‘‘head’’? If so, why? If not, why not? How does the fact that the car part of a list is also called the ‘‘head’’ of a list contribute to this controversy? **(Q1)**

While we are at it: how would you check whether a list *x* is the tail of a list *y*. Do not ponder too long:

```
(require 'headp)
(define (tailp x y)
  (headp (reverse x) (reverse y)))
```

Why is that fact that

```
(tailp () '#whatever) => :t
```

less controversial than the fact that

```
(headp () '#whatever) => :t
```

Bonus exercise: find better names for *headp* and *tailp*.

4.2 find the n'th tail of a list

The *nth* function extracts the tail of a list that starts at the *n*'th element of that list. When the list is too short, it returns **`:f`**:

```
(nth '#0 '#abc) => '#abc
(nth '#1 '#abc) => '#bc
(nth '#2 '#abc) => '#c
(nth '#5 '#abc) => :f
```

Here is the code of *nth*:

```
(require '~nmath)

(define (nth n x)
  (cond ((zero n) x)
        ((null x) :f)
        (t (nth (- n '#1)
                (cdr x)))))
```

When the argument *n* is the same as the the length of the list, *nth* returns an empty list. In particular:

```
(nth '#0 ()) => ()
```

When you swap the first two predicates of the **`cond`** of *nth*, it will return **`:f`** in the above case. Which version do you consider more consistent? Why?

Do you think it is a good idea that the first list member has an offset of `'#0`? Implement a version that starts numbering members at `'#1`. What are the advantages and disadvantages of each version? What will your version return when you pass an offset of zero to it?

4.3 count the atoms of a form

The *count* function counts the atoms of a form recursively:

```
(count ()) => '#0
(count 'a) => '#1
(count '(a (b (c) d) e)) => '#5
```

Here is the code of the *count* function:

```
(require '~nmath)

(define (count x)
  (cond ((null x) '#0)
        ((atom x) '#1)
        (t (+ (count (car x))
              (count (cdr x))))))
```

Strictly speaking, *count* counts only the symbols of a form, because **`()`** is also an atom. If *count* returned `'#1` for `()`, though, its results would be counter-intuitive. Why? **(Q2)**

Count uses structural recursion. Do you think it can be written in a more efficient way?

This is an interesting question. Here is a version of *count* that uses tail calls exclusively:

```
(require '~nmath)

(define (count1 x)
  (letrec
    ((c (lambda (x r s)
          (cond ((null x)
                  (cond ((null s) r)
                        (t (c (car s) r (cdr s)))))
                ((atom x)
                  (c () (+ '#1 r) s))
                (t (c (car x) r (cons (cdr x) s)))))))
    (c x '#0 ())))
```

It uses the internal function *c* that keeps its intermediate result (the atoms counted so far) in the extra parameter *r*. This technique was used earlier in this book to convert linear recursive programs to tail-recursive ones. Can this technique also be applied to turn structural recursion into tail recursion?

The *c* function uses an additional parameter *s* which stores the nodes to be visited in the future. Whenever it finds a new pair, it saves its cdr part in *s* and then starts traversing the car part of that pair.

When *count* finds an instance of `()`, it removes the head of *s* and traverses the structure that was stored there. Of course, the name *s* should invoke the connotation of a ‘‘stack’’, and this is exactly what it is.

The only difference between the structural recursive version and the version using tail calls is that the version using tail calls stores activation records on an explicit stack rather than on zenlisp’s internal stack. The recursion itself cannot be removed. It is inherent in the problem.

This particular case illustrates nicely that attempting to remove recursion from solutions for inherently recursive problems often decreases readability while it barely increases efficiency.

4.4 flatten a tree

A tree is ‘‘flattened’’ by turning it into a flat list that contains the same member in the same order as the original tree:

```
(flatten '((a) (b (c)) (d (e (f))))) => '#abcdef
```

Here is the code of *flatten1*, which performs this operation — although in a highly inefficient way:

```
(define (flatten1 x)
  (cond ((null x) x)
        ((atom x) (list x))
        (t (append (flatten1 (car x))
                   (flatten1 (cdr x))))))
```

Can you name a few reasons why this implementation is far from efficient? (Yes, the obvious structural recursion is only one of them.)

Here is an improved version. It uses a clever trick to avoid structural recursion:

```
(define (flatten x)
  (letrec
    ((f (lambda (x r)
          (cond ((null x) r)
                ((atom x) (cons x r))
                (t (f (car x)
                      (f (cdr x) r)))))))
    (f x ())))
```

The function still applies itself twice, but the result of the first application is passed to the second one, which is a tail call, thereby turning structural recursion into linear recursion.

Can this version of *flatten* be transformed into a version using tail calls, like *count*? Would this transformation improve efficiency any further? **(Q3)**

4.5 partition a list

In the previous part the functions *filter* and *remove* were introduced [page 42]. One of the functions extracts members satisfying a predicate, the other one removes members satifying a predicate.

Can you write a program that combines the functions of *filter* and *remove*? Hint: the function should return two lists.

Here is the code of *partition*, which partitions a list into members satisfying and not satisfying a given predicate:

```
(define (partition p a)
  (letrec
    ((partition3
       (lambda (a r+ r-)
         (cond ((null a)
                 (list r+ r-))
               ((p (car a))
                 (partition3 (cdr a)
                             (cons (car a) r+)
                             r-))
               (t (partition3 (cdr a)
                              r+
                              (cons (car a) r-)))))))
    (partition3 (reverse a) () ())))
```

The first member of its result is equal to the output of *filter* and its second member resembles the output of *remove*. Would it make sense to implement *filter* and *remove* on top of *partition*?

4.6 folding over multiple lists

The fold function (explained on page 45, code on page 271) folds lists to values:

```
(fold (lambda (x y) (list 'op x y)) '0 '(a b c))
=> '(op (op (op 0 a) b) c)
```

The Revised^6 Report on the Algorithmic Language Scheme (R^{6}RS) — although being highly controversial — defines a few interesting functions. One is *fold-left*, which allows to fold over multiple lists:

```
(fold-left (lambda (x y z) (list 'op x y z))
           '0
           '(a b c)
           '(d e f))
=> '(op (op (op 0 a d) b e) c f)
```

All lists must have the same length. Can you implement *fold-left*? Hint: this function has some things in common with **`map`** [page 44] and **`fold`**. In particular, the following functions are helpful when implementing it:

```
(define (car-of a) (map car a))
(define (cdr-of a) (map cdr a))
```

In fact, *fold-left* is exactly like **`fold`**, but uses the techniques of **`map`** to handle variadic arguments:

```
(define (fold-left f b . a*)
  (letrec
    ((fold
       (lambda (a* r)
         (cond ((null (car a*)) r)
               (t (fold (cdr-of a*)
                        (apply f r (car-of a*))))))))
    (cond ((null a*) (bottom 'too-few-arguments))
          (t (fold a* b)))))
```

The R^{6}RS also defines a variant of **`fold-r`** [page 271] that can handle multiple lists. Unsurprisingly it is called *fold-right*. It folds over multiple lists as follows:

```
(fold-right (lambda (x y z) (list 'op x y z))
            '0
            '(a b c)
            '(d e f))
=> '(op a d (op b e (op c f 0)))
```

Fold-right is a bit trickier to write than *fold-left*. Do you want to give it a try before looking at the following code?

```
(define (fold-right f b . a*)
  (letrec
    ((foldr
       (lambda (a* r)
         (cond ((null (car a*)) r)
               (t (foldr (cdr-of a*)
                         (apply f (append (car-of a*)
                                          (list r)))))))))
    (cond ((null a*) (bottom 'too-few-arguments))
          (t (foldr (map reverse a*) b)))))
```

Fold-right uses **`append`** in its general case. Is this a problem? If so, what can you do about it? If not, why not? **(Q4)**

4.7 substitute variables

The *substitute* function replaces variables in forms with values stored in a given environment:

```
(define env '((x . foo) (y . bar)))
(substitute '(cons x y) env) => '(cons foo bar)
```

Substitute is rather straight forward to implement. You may try it before reading ahead. Hint: its code is similar to *replace* [page 30].

```
(define (substitute x env)
  (letrec
    ((value-of
       (lambda (x)
         (let ((v (assq x env)))
           (cond (v (cdr v))
                 (t x)))))
     (subst
       (lambda (x)
         (cond ((null x) ())
               ((atom x) (value-of x))
               (t (cons (subst (car x))
                        (subst (cdr x))))))))
    (subst x)))
```

What happens if you remove the clause **`((null x) ())`** from the **`cond`** of *subst*?

Why can *substitute* not be used to substitute variables with values in the bodies of lambda forms? Hint: this is related to beta reduction [page 59]. **(Q5)**

5. sorting

5.1 insertion sort

The *insert* function inserts an item into an ordered list. An ordered list is a list

$(x_1\ x_2\ \ldots\ x_n)$

where a predicate *p* holds for each two consecutive members:

$(p\ x_i\ x_{i+1})$ `=> :t`

for each *i* in *1..n-1.* Given a variadic predicate *p*,

`(apply` *p x*`)`

applies to each list that is ordered under *p*.

Here is the code of *insert*:

```
(define (insert p x a)
  (letrec
    ((ins
       (lambda (a r)
         (cond ((or (null a) (p x (car a)))
                 (append (reverse (cons x r)) a))
               (t (ins (cdr a) (cons (car a) r)))))))
    (ins a ())))
```

A sorted list is constructed by successively inserting elements into an originally empty list:

```
(load ~nmath)
(insert < '#5 ()) => '(#5)
(insert < '#1 '(#5)) => '(#1 #5)
(insert < '#7 '(#1 #5)) => '(#1 #5 #7)
(insert < '#3 '(#1 #5 #7)) => '(#1 #3 #5 #7)
```

Note that only asymmetric predicates (like **<**) can impose an order on a list. For example, using the (symmetric) **neq** predicate to insert elements to a list does not order that list:

```
(load ~nmath)
(insert neq '#5 ()) => '(#5)
(insert neq '#1 '(#5)) => '(#1 #5)
(insert neq '#7 '(#1 #5)) => '(#7 #1 #5)
(insert neq '#3 '(#7 #1 #5)) => '(#3 #7 #1 #5)
```

A predicate is symmetric if **(***p b a***)** follows from **(***p a b***)** for each *a* and *b*. A predicate is asymmetric if this implication does not hold. Asymmetric predicates that are often used to order lists include **<**, **<=**, **>**, and **>=**.

The *isort* function inserts multiple members into an originally empty list, thereby effectively sorting those members:

```
(isort < '(#5 #1 #7 #3)) => '(#1 #3 #5 #7)
```

Here comes its code:

```
(require 'insert)

(define (isort p a)
  (Letrec
    ((sort
       (lambda (a r)
         (cond ((null a) r)
               (t (sort (cdr a)
                        (insert p (car a) r)))))))
    (sort a ())))
```

This algorithm is widely known as *insertion sort*.

How many times does *isort* have to apply its predicate *p* in order to sort a list of **n** elements, if:

– the elements are sorted in reverse order;
– the elements already are sorted;
– the elements are randomly distributed?

Do the above values differ by a wide margin? Does the result make insertion sort a practicable sorting algorithm or not? **(Q6)**

5.2 quicksort

Quicksort is probably one of the most efficient algorithms around. It uses a *divide and conquer* approach. This approach works by dividing the problem into smaller subproblems, solving them, and then re-assembling the result: [10]

```
(require 'partition)

(define (quicksort p a)
  (letrec
    ((sort
       (lambda (a)
         (cond ((or (null a) (null (cdr a))) a)
               (t (let ((p* (partition (lambda (x) (p (car a) x))
                                       (cdr a))))
                    (append (sort (cadr p*))
                            (list (car a))
                            (sort (car p*)))))))))
    (sort a)))
```

10 The approach works even in the real world. By separating the individuals of big masses of people, the people can easily be controlled by small, power-hungry groups. Advertisements and “news” are typical real-world divide and conquer tools. What can we do about them?

When quicksort sorts a list, it first divides that list into smaller lists, one containing members below a given threshold and one containing members above that threshold. The **`trace`** meta function can be used to look under the hood:

```
(trace sort) => :t
(quicksort < '(#5 #1 #9 #3 #7))
+ (sort (#5 #1 #9 #3 #7))
+ (sort (#1 #3))          ; sort members <5
+ (sort ())               ; sort members <5 and <1
+ (sort (#3))             ; sort members <5 and >1
+ (sort (#9 #7))          ; sort members >5
+ (sort (#7))             ; sort members >5 and <9
+ (sort ())               ; sort members >5 and >9
=> '(#1 #3 #5 #7 #9)
```

After sorting all partitions, *quicksort* re-assembles them using **`append`**:

```
(trace append) => :t
(quicksort < '(#5 #1 #9 #3 #7))
+ (append () (#1) (#3))
+ (append (#7) (#9) ())
+ (append (#1 #3) (#5) (#7 #9))
=> '(#1 #3 #5 #7 #9)
```

Note that *quicksort* itself never applies the predicate *p*. The list is sorted by partitioning it again and again:

```
(partition (lambda (x) (p '#5 x))
           '(#1 #9 #3 #7))
=> '((#9 #7) (#1 #3))
```

After partitioning the list `'(#1 #9 #3 #7)`, the first partition contains only values below 5 and the second one only values above 5. These partitions are sorted recursively and then concatenated (together with the theshold value iself).

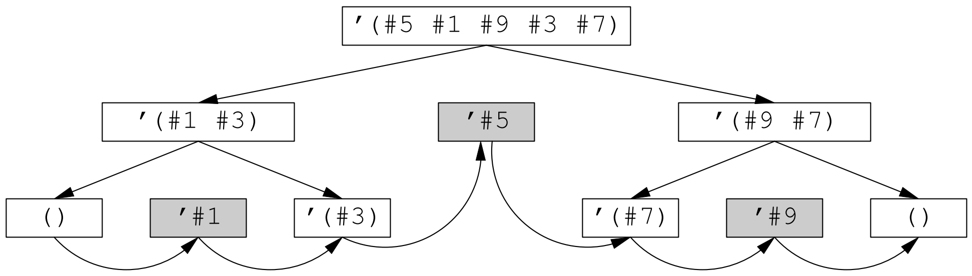

Fig. 1 – divide and conquer approach

Figure 1 illustrates the divide and conquer paradigm. The original list is recursively broken down into smaller lists until the smaller lists are trivial to sort, because they are either empty or contain only single values. The straight arrows indicate partitioning and the grey boxes contain threshold values. The curved arrows denote the **`append`** operations.

How does *quicksort* compare to *isort*? How does *quicksort* perform when sorting already sorted and reverse sorted input?

5.3 mergesort

The performance of both *quicksort* and *isort* depends on the data passed to them. This is, of course, not a desirable property for a sorting algorithm.

Mergesort is about as efficient as *quicksort* while its performance is constant. In environments without mutable data (like zenlisp), *mergesort* is even slightly more efficient than *quicksort*.

Here is its code:

```
(define (mergesort p a)
  (letrec
    ((split
       (lambda (a r1 r2)
         (cond ((or (null a)
                    (null (cdr a)))
                 (list (reverse r2) r1))
               (t (split (cddr a)
                         (cdr r1)
                         (cons (car r1) r2))))))
     (merge
       (lambda (a b r)
         (cond
           ((null a)
             (cond ((null b) r)
                   (t (merge a (cdr b) (cons (car b) r)))))
           ((null b)
             (merge (cdr a) b (cons (car a) r)))
           ((p (car a) (car b))
             (merge a (cdr b) (cons (car b) r)))
           (t (merge (cdr a) b (cons (car a) r))))))
     (sort
       (lambda (a)
         (cond ((or (null a)
                    (null (cdr a)))
                 a)
               (t (let ((p* (split a a ())))
                    (merge (reverse (sort (car p*)))
                           (reverse (sort (cadr p*)))
                           ()))))))))
    (sort a)))
```

Mergesort's performance is constant because it *splits* its input rather than partitioning it. This is what the *split* function inside of *mergesort* does:

```
(split '#abcdef '#abcdef ()) => '(#abc #def)
```

As this example shows, the list is simply split in the middle, no matter which members it contains. Therefore *mergesort* always divides its input in an optimal way.

You may wonder why *split* actually splits the list in the middle. Would it not be easier to split the list by simply pushing its members to two other lists alternately? Like this:

```
(define (naive-split a r1 r2)
  (cond ((null a)
          (list r1 r2))
        ((null (cdr a))
          (list (cons (car a) r1) r2))
        (t (naive-split (cddr a)
                        (cons (car a) r1)
                        (cons (cadr a) r2)))))
(naive-split '#abcdef () ()) => '(#eca #fdb)
```

While this function is a bit simpler and possibly even a bit more efficient than *split*, it is also wrong. Can you explain why?

The answer becomes obvious when splitting less trivial data:

```
(define set '((#5 b) (#1 a) (#5 c) (#7 a) (#9 a) (#3 a)))
(split set set ())
=> '(((#5 b) (#1 a) (#5 c)) ((#7 a) (#9 a) (#3 a)))
(naive-split set () ())
=> '(((#9 a) (#5 c) (#5 b)) ((#3 a) (#7 a) (#1 a)))
```

Each member of the above set contain a numeric key and an additional symbol. Both splitting functions split the set evenly, but *split* preserves the order of the members while *naive-split* swaps some members.

While both splitting functions would work fine in *mergesort*, the naive variant would result in a sorting function that is *unstable.* An unstable sorting function sorts members just fine, but may swap members with equal keys in the process. Stability is a desirable property in sorting functions.

BTW, both *quicksort* and *mergesort* are only stable when non-strict predicates like **<=** and **>=** are used. They are unstable when strict predicates like **<** and **>** are passed to them. *Isort* is only stable when strict predicates are used.

Which of these two approaches do you consider advantageous? Why? **(Q7)**

Can you modify *isort* in such a way that it becomes stable when using non-strict predicates? **(Q8)**

You probably saw it coming: can you modify *quicksort* and *mergesort* in such a way that they become stable when using strict predicates? **(Q8)**

5.4 unsorting lists

When examining sorting functions, it would be nice to have some unsorted data. The *unsort* function creates lists of unsorted natural numbers:

```
(unsort '(#1 #2 #3 #4 #5 #6 #7 #8 #9 #10) '#3)
=> '(#8 #5 #1 #7 #10 #9 #3 #2 #6 #4)
```

Its input must be a list of consecutive natural numbers, but they need not occur in any specific order:

```
(unsort '(#8 #5 #1 #7 #10 #9 #3 #2 #6 #4) '#5)
=> '(#5 #8 #1 #2 #3 #6 #7 #4 #10 #9)
```

The second argument of *unsort* is the "seed" — the first member of the list that will be unsorted. It must be a natural number that is less than the length of the given list.

Here is the code of *unsort*:

```
(require '~nmath)
(require 'nth)

(define (unsort a seed)
  (letrec
    ((remove-nth
       (lambda (a n r)
         (cond ((zero n)
                 (cond ((null a) (reverse r))
                       (t (append (cdr a) (reverse r)))))
               (t (remove-nth (cdr a)
                              (- n '#1)
                              (cons (car a) r))))))
     (unsort4
       (lambda (a n k r)
         (cond ((zero k) (cons (car a) r))
               (t (unsort4 (remove-nth a n ())
                           (remainder (car a) k)
                           (- k '#1)
                           (cons (car (nth n a)) r)))))))
    (unsort4 a seed (- (length a) '#1) ())))
```

When combined with *iota* [page 86], *unsort* can be used to create larger sets of unsorted numbers. Such sets are useful for examining the efficiency of sorting functions. The following definitions create some test sets:

```
(define sorted-set (iota '#1 '#100))          ; 1..100
(define reverse-set (reverse sorted-set))     ; 100..1
(define random-set (unsort sorted-set '#99))  ; random order
```

In combination with the **`stats`** meta function, sets like these can be used to test how sorting functions perform when sorting already sorted input, reverse sorted input and random input.

Figure 2 summarizes some results. The numbers in the table denote *reduction steps*. Each reduction, like the retrieval of the value of a variable or a function application, is one step.

For each algorithm, the steps needed to sort already sorted data, reverse sorted data, and random data are given. The fact that *quicksort* performs dramatically bad when sorting non-random data is in fact a flaw of the implementation, not a flaw of the algorithm.

	isort			quicksort			mergesort		
size	**sorted**	**reverse**	**random**	**sorted**	**reverse**	**random**	**sorted**	**reverse**	**random**
10	43,764	9,701	**29,923**	38,014	46,245	**30,587**	18,705	17,679	**22,619**
20	174,833	20,422	**95,433**	158,906	184,724	**55,261**	46,805	45,262	**65,116**
30	408,957	32,096	**244,376**	371,783	430,858	**126,311**	79,393	82,692	**109,486**
40	758,136	44,723	**346,366**	685,945	796,647	**169,333**	120,065	118,086	**170,110**
50	1,234,370	58,303	**680,393**	1,110,692	1,294,091	**256,190**	168,093	166,374	**235,366**
60	1,849,659	72,836	**984,197**	1,655,324	1,935,190	**422,885**	208,201	219,251	**310,940**
70	2,616,003	88,322	**1,388,878**	2,329,141	2,731,944	**434,277**	269,668	274,136	**390,619**
80	3,545,402	104,761	**1,883,191**	3,141,443	3,696,353	**621,906**	324,707	324,210	**477,393**
90	4,649,856	122,153	**2,516,138**	4,101,530	4,840,417	**623,504**	399,655	394,382	**569,448**
100	5,896,572	140,126	**3,137,972**	5,193,349	6,131,343	**825,283**	457,256	460,812	**676,687**

Fig. 2 – run times of sorting algorithms

The fact that the present implementation of *quicksort* performs as bad as *isort* when sorting already sorted data and equally bad when sorting reverse sorted data can be remedied by selecting a random threshold from the current partition (*p**).

Nevertheless the Quicksort algorithm *does* have a weakness that is caused by its dependence on the data to sort. Much research has been done in order to break this dependency, and in fact *efficient* implementations of Quicksort are the fastest sorting functions in practice.

Mergesort is much easier to implement efficiently than Quicksort because its performance does not depend on the data to sort [11] and it has an advantage in purely functional environments where data may not be mutated.

It is also worth noting that Insertion Sort, which performs abysmally on random and already-sorted sets, excels at sorting sets in reverse sorted order. In this case, it is even more efficient than the other two algorithms. In fact, the following algorithm is more suitable for inserting *small* amounts of data into a sorted list than the other two sorting algorithms:

```
(define (insert-small-set p small sorted)
  (isort p (append small (reverse sorted))))
```

For larger sets, *mergesort* should be used in the place of *isort* and **`reverse`** should be omitted. Feel free to explore reasonable limits for the size of the set to insert before switching to Mergesort becomes imperative.

When looking at the columns that compare the steps needed to sort random data (printed in

11 However, *mergesort* exhibits a slight bias toward sorted data in the table. Can you explain this?

boldface characters in figure 2), you may notice that individual numbers say little about the qualities of the sorting algorithms. It is the *development* of these numbers as the sizes of the input sets increase which is interesting.

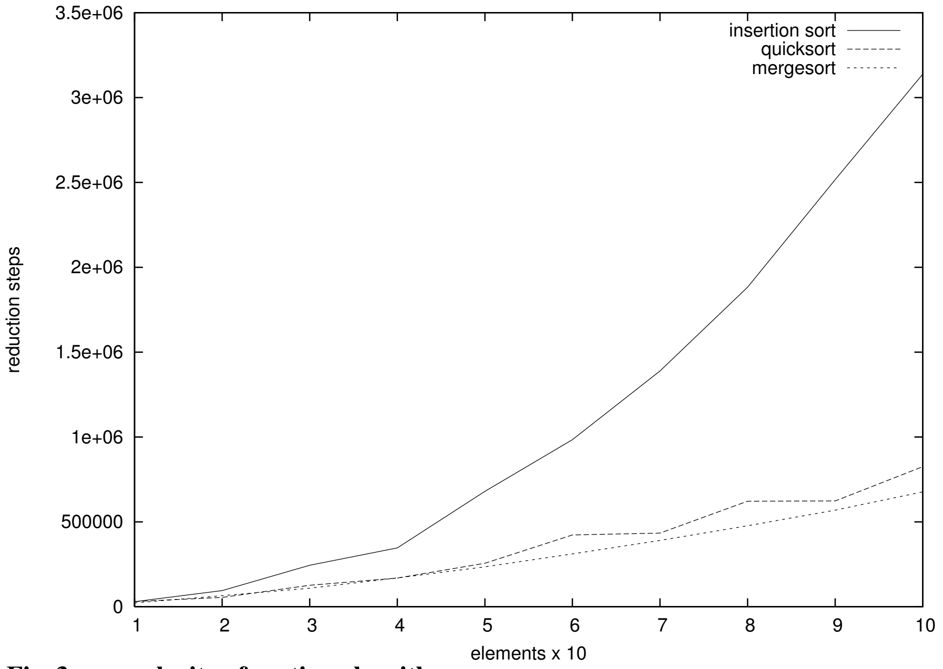

Fig. 3 – complexity of sorting algorithms

This development is called the *complexity* of a function. Figure 3 renders the complexity of the sorting functions discussed in this section as graphs. The *x*-axis reads the size of the random sets to sort, the *y*-axis the steps needed to sort a given set.

As can be seen in the graph, the run time of Insertion Sort grows quickly when the set size increases, while the run times of both Quicksort and Mergesort grow only slowly. Flat curves indicate high efficiency, steep curves indicate bad performance.

Complexity can be expressed without using curves by using the so-called *big-O notation*: $O(n \times 5)$ may be interpreted as "the runtime of the described algorithm multiplies by five when increasing its input by 1". Big-O notation may describe both space and time requirements of an algorithm. In this section, we concentrate on time.

The average (time) complexity of Insertion Sort is $O(n^2/2)$, and the (average) complexity of Quicksort and Mergesort is $O(n \times log(n))$. As we have seen, Quicksort has a worst-case performance of $O(n^2/2)$ (like Insertion sort), while the worst-case performance of Mergesort is (about) equal to its average performance.

The important part of the big-O notation is the formula itself, not their coefficients. For example, $O(n \times 10000)$ is generally *much* better than $O(1.1^n)$, even though the former has a much greater coefficient. For small values of *n*, the latter is indeed more efficient:

$O(10 \times 10000) = 100000$
$O(1.1^{10}) = 2.5$

At $n=150$, they almost break even:

$O(150 \times 10000) = 1500000$
$O(1.1^{150}) = 1617717.8$

But at $n=1000$, things look entirely different:

$O(1000 \times 10000) = 100000000$
$O(1.1^{1000}) = 2469932918005826334124088385085221477709733$

$O(c \times n)$ (where c is constant) describes ''linear'' complexity, which is ''very good'', while $O(c^n)$ indicates ''exponential'' complexity, which in most cases means that either the algorithm it describes is close to unusable or the problem it attempts to solve is *very* hard.

Figure 4 provides an overview over different classes of complexity, from best to worst.

formula	**name**
$O(c)$	constant
$O(c \times log(n))$	logarithmic
$O(c \times n)$	linear
$O(c \times n^2)$	quadratic
$O(n^c)$	polynomial or geometric
$O(c^n)$	exponential

Fig. 4 – classes of complexity

BTW: which complexity does the *unsort* function have in the average case and in the worst case? **(Q9**)

Do you think it can be improved?

6. logic and combinatoric functions

6.1 turning lists into sets

A *set* is a list of unique members. For instance, `'(a b c)` is a set, but `'(a a b)` is not because `'a` occurs twice in the list. Just as a finger exercise: can you create a tail recursive function that converts a list to a set? E.g.:

```
(list->set '(a a b)) => '(a b)
```

Here is one possible solution:

```
(define (list->set a)
  (letrec
    ((l->s
       (lambda (a r)
         (cond ((null a)
                 (reverse r))
               ((member (car a) r)
                 (l->s (cdr a) r))
               (t (l->s (cdr a) (cons (car a) r)))))))
    (l->s a ())))
```

6.2 union and intersection of sets

Using *list->set* computing the union of multiple sets is so simple that it is barely worth writing a function for it:

```
(require 'list-to-set)
```

```
(define (union . a)
  (list->set (apply append a)))
```

The intersection of sets is a bit trickier. It was discussed in depth earlier in this book [page 24]. Here comes a tail recursive variant using **`fold`**.

Of couse you may try writing it yourself before reading ahead.

```
(define (intersection . a)
  (letrec
    ((intersection3 (lambda (a b r)
      (cond ((null a)
              (reverse r))
            ((member (car a) b)
              (intersection3 (cdr a) b (cons (car a) r)))
            (t (intersection3 (cdr a) b r))))))
    (fold (lambda (a b)
            (intersection3 a b ()))
          (car a)
          a)))
```

6.3 find members with a given property

The *any* function checks whether a list contains at least one member with a given property:

```
(define (any p a)
  (cond ((null a) :f)
        (t (or (p (car a))
               (any p (cdr a))))))
(any atom '(#ab #cd e)) => :t
```

The R^6RS defines a generalized version of *any* called *exists*. It relates to *any* in the same way as *map-car* [page 44] relates to **`map`**: it accepts any positive number of list arguments and an *n*-ary function to be applied to them.

The following application of *exists* finds out whether there exists any position **i** in the lists **a**, **b**, **c** where $\mathbf{b_i}$ is the least value of the triple *(*$\mathbf{a_i}$,$\mathbf{b_i}$,$\mathbf{c_i}$*)*.

```
(exists (lambda (ai bi ci)
          (and (< bi ai)
               (< bi ci)))
        a
        b
        c)
```

Once again the *car-of* and *cdr-of* functions [page 67] prove helpful when turning a fixed-argument function into a variadic one:

```
(define (exists p . a*)
  (letrec
    ((exists*
       (lambda (a*)
         (cond ((null (car a*)) :f)
               (t (or (apply p (car-of a*))
                      (exists* (cdr-of a*))))))))
    (exists* a*)))
```

BTW: *exists* is not a predicate. Can you explain way?

Exists can do something more interesting than checking whether a tuple with the given property exists: it can return that tuple:

```
(exists (lambda (ai bi ci)
          (and (< bi ai)
               (< bi ci)
               (list ai bi ci)))
        '(#3 #1 #4 #1 #5)
        '(#9 #2 #6 #5 #3)
        '(#5 #8 #9 #7 #9))
=> '(#5 #3 #9)
```

Because *exists* may return a value other than **`:t`** or **`:f`** when a non-predicate is passed to it, it is itself not a predicate.

6.4 verify properties

You may have observed that the *exists* function implements the *extistential quantor*:

There exists an **x**, **y**, *... for which (***p x y** *...).*

The R^{6}RS also describes a function implementing the *universal quantor* (for all **x**, **y**, ...). Unsurprisingly it is called *for-all* and returns truth if the given predicate applies to *all* tuples that are formed by combining list elements at equal positions:

```
(for-all eq '#abcdef '#abcdef) => :t
```

When a non-predicate is passed to *for-all*, it returns the *last* tuple that could be formed (or **:f**):

```
(for-all (lambda (x y)
           (and (eq x y)
                (list x y)))
         '#abcdef
         '#abcdef) => #ff
```

Here is the code of *for-all*:

```
(define (for-all p . a*)
  (letrec
    ((forall*
       (lambda (a*)
         (cond ((null (car a*)) :t)
               ((null (cdar a*))
                 (apply p (car-of a*)))
               (t (and (apply p (car-of a*))
                       (forall* (cdr-of a*))))))))
    (forall* a*)))
```

In which way does the behavior of the function change when you omit the clause

```
((null (cdar a*)) (apply p (car-of a*)))
```

from the above code? **(Q10)**

For-all returns **:t** when empty lists are passed to it and *exists* returns **:f** in this case. Do you think this makes sense? Why?

6.5 combinations of sets

The functions discussed in this section are a bit more complex, so if you need a break, this would be an excellent time to take one.

A *combination* of the **n**'th order (an **n**-combination) of a source set **a** is an **n**-element set of elements from **a**. For example, `'#ab` is a 2-combination of the set `'#abc`.

The *combine* function generates *all* possible **n**-element combinations without repetition from a given set:

```
(combine '#2 '#abc) => '(#ab #ac #bc)
```

*Combine** creates combinations *with* repetition:

```
(combine* '#2 '#abc) => '(#aa #ab #ac #bb #bc #cc)
```

Because the algorithms for these functions are very similar, they are both implemented in the higher-order *combine3* function. The differing part of the algorithms is passed to *combine3* as an argument.

Combinations of **n** elements with repetition are created as follows:

If **n**=0, the set of combinations is empty:

```
(combine '#0 x) => () ; for any x
```

If **n**=1,

```
(combine '#1 x) —> (map list x) ; for any x
```

For **n**>1, combinations are created recursively. The first step in this process is to create all possible non-empty tails of the source set. This is what the *tails-of* function does:

```
(define (tails-of set)
  (cond ((null set) ())
        (t (cons set (tails-of (cdr set))))))

(tails-of '#abcd) => '(#abcd #bcd #cd #d)
```

N-combinations with repetition are created from the result of *tails-of* as follows: the head of each sublist is attached to all **n**-*1*-element combinations of the same sublist. The following example outlines this process for **n**=2 and **set**=`'#abcd`:

head	**tail**	**1-combinations**	**result**
`#a`	`#abcd`	`(#a #b #c #d)`	`(#aa #ab #ac #ad)`
`#b`	`#bcd`	`(#b #c #d)`	`(#bb #bc #bd)`
`#c`	`#cd`	`(#c #d)`	`(#cc #cd)`
`#d`	`#d`	`(#d)`	`(#dd)`

The concatenation of the **result** column is the set of all 2-combinations of `'#abcd`:

```
'(#aa #ab #ac #ad #bb #bc #bd #cc #cd #dd)
```

In the above example, *combine* calls itself with **n**=1, so this application is handled by a trivial case. Here is what happens when combinations of a higher order are generated (the example creates the 3-combinations of `'#abcd`):

head	tail	2-combinations	result
#a	#abcd	(#aa #ab #ac #ad #bb #bc #bd #cc #cd #dd)	(#abc #abd #acc #acd #add #aaa #aab #aac #aad #abb)
#b	#bcd	(#bb #bc #bd #cc #cd #dd)	(#bbb #bbc #bbd #bcc #bcd #bdd)
#c	#cd	(#cc #cd #dd)	(#ccc #ccd #cdd)
#d	#d	(#dd)	(#ddd)

The 2-combinations that are appended to the head of each tail set are created in the same way as in the previous example. Because **n** decreases, the trivial case is finally reached.

Here follows the code of the *combine2* function which creates combinations with repetition:

```
(require '~nmath)

(define (combine2 n set)
  (cond
    ((zero n) ())
    ((one n) (map list set))
    (t (apply
         append
         (map (lambda (tail)
                (map (lambda (sub)
                       (cons (car tail) sub))
                     (combine2 (- n '#1) tail)))
              (tails-of set))))))
```

The general case of the function contains two nested **map**s which create combinations recursively. The list passed to the outer **map** is the tail set of the current source set. The inner **map** recurses to create combinations of a lower order and then attaches these combinations to the head of *tail*. Finally, **apply**ing **append** flattens the result of the outer map.

Combinations without repetition are created in basically the same way as combinations with repetition. The only difference is that items from the source set may not be reused, so they have to be removed from the set before creating lower-order combinations. Here is how it works:

head	tail	1-combinations	result
#a	#bcd	(#b #c #d)	(#ab #ac #ad)
#b	#cd	(#c #d)	(#bc #bd)
#c	#d	(#d)	(#cd)

So instead of the entire set only the cdr part should be passed to *combine2* when recursing. The modification is so trivial that the function doing the cdr operation is passed to *combine3* as an additional argument. Differences to *combine2* are rendered in boldface characters:

```
(define (combine3 n set rest)
  (lambda (n set)
    (cond
      ((zero n) ())
      ((one n) (map list set))
      (t (apply
           append
           (map (lambda (tail)
                  (map (lambda (sub)
                         (cons (car tail) sub))
                       (combine3 (- n '#1) (rest tail) rest)))
                (tails-of set)))))))
```

When *rest*=**`cdr`** is passed to *combine3*, it computes combinations without repetition. To compute combinations with repetition, the identity function **`id`** [12] is passed to it intead. So *combine* and *combine** can be defined as follows:

```
(define (combine n set)
  (combine3 n set cdr))

(define (combine* n set)
  (combine3 n set id))
```

Which kind of recursion does *combine3* use?

Do the complexities of *combine* and *combine** differ? Estimate their complexities. **(Q11)**

6.6 permutations of sets

A *permutation* differs from a *combination* in the respect that order does not matter in combinations but does matter in permutations. Hence the sequences `'#ab` and `'#ba` denote the same combination but different permutations. Both `'#ab` and `'#ba` *combine* `a` with `b`, but `'#ab` is a permutation of `'#ba` (in this case the only one).

Like combinations, permutations can be created with and without repetition. The *permute* and *permute** functions introduced in this section create all possible permutations of a set with and without repetition:

```
(permute '#2 '#abc) => '(#ab #ba #ac #ca #bc #cb)
(permute* '#2 '#abc) => '(#aa #ab #ac #ba #bb #bc #ca #cb #cc)
```

We will start with a function that creates **n**-permutations without repetition from source sets of the size **n**, e.g.:

```
(permutations '#abc) => '(#abc #acb #bca #bac #cab #cba)
```

The trivial cases of this function are simple. 0-permutations are empty:

12 **`Id`** is defined as **`(lambda #x x)`**.

```
(permutations ()) => ()
```

and 1-permutation are equal to a set containin the source set:

```
(permutations '(x)) —> (list '(x)) ; for any atom x
```

Permutations are created in almost the same was as combinations, but because order does matter in permutations, each of the elements of the source set has to occupy each position once. This is easily achieved by rotating the members of the set:

```
'#abcd
'#bcda
'#cdab
'#dabc
```

The *rotate* function rotates a set by one position:

```
(define (rotate x)
  (append (cdr x) (list (car x))))
```

Using *rotate*, creating *all* rotations is easy:

```
(define (rotations x)
  (letrec
    ((rot (lambda (x n)
            (cond ((null n) ())
                  (t (cons x (rot (rotate x)
                                  (cdr n))))))))
    (rot x x)))

(rotations '#abcd) => '(#abcd #bcda #cdab #dabc)
```

The *permutations* function itself looks pretty much like *combine3* [page 83]. It just has different trivial cases and uses *rotations* in the place of *tails-of*:

```
(define (permutations set)
  (cond
    ((null set) ())
    ((null (cdr set)) (list set))
    (t (apply append
              (map (lambda (rotn)
                     (map (lambda (x)
                            (cons (car rotn) x))
                          (permutations (cdr rotn))))
                   (rotations set))))))
```

While this function works fine, it can only create permutations whose order equal to the size of the source set. Permutations of a lower order can be computed by creating the combinations of the same order first and then permutating and appending the results:

```
(combine '#2 '#abc) => '(#ab #ac #bc)
(map permutations '(#ab #ac #bc)) => '((#ab #ba) (#ac #ca) (#bc #cb))
```

```
(apply append '((#ab #ba) (#ac #ca) (#bc #cb)))
  => '(#ab #ba #ac #ca #bc #cb)
```

This is exactly what the following implementation of *permute* does:

```
(require 'combine)

(define (permute n set)
  (apply append (map permutations (combine n set))))
```

Permutations with repetition are created in a similar way as combinations with repetition (see *combine2*, page 82). The only difference is that the whole source set is passed along in each **`map`**:

```
(define (permute* n set)
  (cond
    ((zero n) ())
    ((one n) (map list set))
    (t (apply append
                 (map (lambda (x)
                        (map (lambda (sub)
                               (cons x sub))
                             (permute* (- n '#1) set)))
                      set)))))
```

What complexity does the *permutations* function have?

The 2-permutations of a 2-element set are equal to the rotations of that set:

```
(permutations '#xy) => '(#xy #yx)
   (rotations '#xy) => '(#xy #yx)
```

So we could add a third trivial case to the *permutations* function:

```
    ((null (cddr set)) (rotations set))
```

Where would you insert this case? Would this modification make sense? Would it improve the run time of *permutations*? Would it change its complexity? **(Q12)**

7. math functions

7.1 sequences of numbers

The *iota* function creates sequences of integer numbers:

```
(iota '#1 '#10) => '(#1 #2 #3 #4 #5 #6 #7 #8 #9 #10)
(iota '#-3 '#-1) => '(#-3 #-2 #-1)
```

In combination with **map** it can be used to create various kinds of sequences:

```
(map (lambda (x) (* x x))
     (iota '#1 '#5))
=> '(#1 #4 #9 #16 #25)
(require '~combine)
(map (lambda (x) (combine x '#abcde))
     (iota '#0 '#5))
=> '(()
     (#a #b #c #d #e)
     (#ab #ac #ad #ae #bc #bd #be #cd #ce #de)
     (#abc #abd #abe #acd #ace #ade #bcd #bce #bde #cde)
     (#abcd #abce #abde #acde #bcde)
     (#abcde))
(map (lambda (x) (length (combine* x '#abcd)))
     (iota '#1 '#10))
=> '(#4 #10 #20 #35 #56 #84 #120 #165 #220 #286)
```

The implementation of *iota* is much simpler than many of its applications:

```
(require '~imath)

(define (iota lo hi)
  (letrec
    ((j (lambda (x r)
          (cond ((< x lo) r)
                (t (j (- x '#1) (cons x r)))))))
    (j (integer hi) ())))
```

Can you use *iota* to estimate the complexities of some functions like *combine**, *permute**, *unsort*, or *iota* itself?

7.2 fast factorial function

The *factorial* function computes **n**! ("**n** factorial"), or

```
(* 1 2 ... n)
```

However, it uses an algorithm that is more efficient than just multiplying the numbers in sequence. It can compute values like 100! in reasonable time, even on a purely symbolic system like zenlisp.

The algorithm is called *recursive product*.

The code follows immediately. Can you see why it is more efficient than the naive approach?

```
(require '~nmath)

(define (factorial n)
  (letrec
    ((r* (lambda (n m)
           (cond ((< m '#2) n)
                 (t (let ((q (quotient m '#2)))
                      (* (r* n q)
                         (r* (+ n q) (- m q)))))))))
    (r* '#1 (natural n))))
```

The *factorial* function is *fat recursive*: it has the same complexity as a structural recursive function, although it does not process a recursive structure. This is generally considered a bad idea, but in this particular case, it actually improves the run time of the function.

Factorial uses a divide and conquer approach [see page 70]. Given the value **n**, it computes the products of *1*..**n**/*2* and **n**/*2*+*1*..**n** first and then multiplies them. It uses the same number of multiplications as the ''naive'' function (one less in fact), but it avoids to multiply large numbers as long as possible, as can be seen in the following table:

naive approach **recursive product**

```
(f '#10)                 (factorial '#10)
+ (* #1 #1)              + (* #1 #2)
+ (* #2 #1)              + (* #4 #5)
+ (* #3 #2)              + (* #3 #20)
+ (* #4 #6)              + (* #2 #60)
+ (* #5 #24)             + (* #6 #7)
+ (* #6 #120)            + (* #9 #10)
+ (* #7 #720)            + (* #8 #90)
+ (* #8 #5040)           + (* #42 #720)
+ (* #9 #40320)          + (* #120 #30240)
+ (* #10 #362880)
```

The output is created by tracing `*` while computing 10*!*.[13]

There is an even simpler (and interestingly more efficient) method for computing **n***!* in zenlisp. This method does not use recursion at all. Hint: it does use a function from this section. **(Q13)**

13 The naive function is: `(define (f x) (cond ((zero x) '#1) (t (* x (f (- x '#1)))))).`

7.3 integer factorization

The *factors* function computes the constituent prime factors of a given integer:

```
(factors '#123456789) => '((#3803 #1) (#3607 #1) (#3 #2))
```

It returns a list of two-element lists where each sublist should be read as car^{cadr}, so the above integer factors into

$$3383^1 \times 3607^1 \times 3^2 \quad = \quad 3383 \times 3607 \times 3 \times 3$$

Factors may take a while to complete when computing the factors of primes, coprimes, or integers composed of rather large factors, like the above. It actually attempts to divide a given number **n** by {*2,3,5,...,sqrt(***n***)*} and memorizes the quotients found:

```
(require '~nmath)

(define (factors n)
  (letrec
    ((quotient+exponent
       (lambda (n m)
         (letrec
           ((div (lambda (n m r)
                   (let ((qr (divide n m)))
                     (cond ((zero (cadr qr))
                             (div (car qr) m (+ '#1 r)))
                           (t (cons n r)))))))
           (div n m '#0))))
     (add-expt
       (lambda (b e r)
         (cond ((zero e) r)
               (t (cons (list b e) r)))))
     (factorize
       (lambda (n d r)
         (let ((lim (sqrt n)))
           (letrec
             ((factorize3
                (lambda (n d r)
                  (let ((rest/exp (quotient+exponent n d)))
                    (let ((q (car rest/exp))
                          (e (cdr rest/exp)))
                      (cond
                        ((< q '#2) (add-expt d e r))
                        ((> d lim) (add-expt n '#1 r))
                        (t (factorize3
                             q
                             (cond ((= d '#2) '#3)
                                   (t  (+ d '#2)))
                             (add-expt d e r)))))))))
             (factorize3 n d r))))))
    (cond
      ((< n '#1) (bottom 'operand-not-positive n))
      ((= n '#1) '#1)
      (t (factorize n '#2 ())))))
```

The function is pretty straight-forward. Its helper function *quotient+exponent* returns the quotient that remains when dividing **n** by **m** a given number of times. The number of divisions is returned as the ‘‘exponent’’ part. For example:

```
(quotient+exponent '#24 '#2) => '(#3 . #3)
```

24 can be divided by 2 three times, so 3 is returned as exponent in the cdr part of the result. $24/2^3 = 3$, so 3 is also returned as quotient in the car part.

```
(quotient+exponent '#24 '#3) => '(#8 . #1)
```

24 can be divided by 3 one time, leaving a quotient of 8.

```
(quotient+exponent '#24 '#9) => '(#24 . #0)
```

24 cannot be divided by 9 at all (zero times, leaving a ‘‘quotient’’ of 24).

Add-expt adds a factor to the result, but only if the exponent is non-zero.

Factorize computes the limit of *sqrt(***n***)* and then iterates through the list of possible divisors.

Can you imagine why the square root of the integer to factor is chosen for the upper limit?

There does not seem to be a really efficient algorithm for factorizing large integers. If there was one, computer-based cryptology might very well become a lost art. This is not meant to stop you from searching for a better algorithm, though.

7.4 partitioning integers

A (number-theoretic) *partition* of an integer **n** is a sum of integers that adds up to **n**. For instance, these are the partitions of 4:

4   3 + 1   2 + 2   2 + 1 + 1   1 + 1 + 1 + 1

Creating partitions is similar to creating permutations. The only difference is that elements of a partition may be split in two, e.g. 3 may be split into 2 and 1, and 2 may be split into 1 and 1.

The following code is based on *permutations* [page 84]. Instead of attaching the head of a set to each permutation of its rest, it attaches each value **i** of the interval *1*..**n** to the partitions of **n**-**i**, e.g.:

for n=4
– 1 is attached to the partitions of 3;
– 2 is attached to the partitions of 2;
– 3 is attached to the partitions of 1;
– 4 is attached to the partitions of 0.

The trivial cases handle the values of zero (giving the empty partition) and one (giving a partition of 1). Partitions are represented as lists, so the partitions of 4 would be written as

`'((#4) (#3 #1) (#2 #2) (#2 #1 #1) (#1 #1 #1 #1)).`

Here is the code (but feel free to try to develop it on your own before reading ahead):

```
(require '~nmath)
(require 'iota)

(define (part n)
  (cond
    ((zero n) '(()))
    ((one n) '((#1)))
    (t (apply append
              (map (lambda (x)
                     (map (lambda (p) (cons x p))
                          (part (- n x))))
                   (iota '#1 n))))))
```

Let us see how it performs:

```
(part '#4)
=> '((#1 #1 #1 #1) (#1 #1 #2) (#1 #2 #1) (#1 #3) (#2 #1 #1) (#2 #2) (#3 #1) (#4))
```

Looks fine except that the list is in reverse order (which is easy to fix) and that it contains some duplicates (in boldface characters). So maybe we need a solution that is based on combinations rather than permutations? Before we dive to deep into this idea: is there something obvious about the above result?

Well?

Exactly: all the non-duplicate partitions are in descending order (but not in *strict* descending order), so we can filter them. This is what the *make-partitions* function does:

```
(define (make-partitions n)
  (letrec
    ((filter-descending
       (lambda (p)
         (cond ((null (cdr p)) p)
               ((apply >= (car p))
                 (cons (car p) (filter-descending (cdr p))))
               (t (filter-descending (cdr p)))))))
    (reverse (filter-descending (part n)))))
```

And indeed:

```
(make-partitions '#4) => '((#4) (#3 #1) (#2 #2) (#2 #1 #1) (#1 #1 #1 #1))
```

Why are we not using *filter* [page 42] to filter the results? **(Q14)**

Give a reasonable upper limit for the number of partitions of 100. Why is *make-partitions* not such a great help in this process?

7.5 exploring the limits of computability

Have a look at the following function:

```
(define (s x) (+ '#1 x))
```

The *s* function is much like **`succ`** [page 276], but increments whole natural numbers instead of just single digits:

```
(s '#0) => '#1
(s (s '#0)) => '#2
(s (s (s '#0))) => '#3
```

Using this function, the addition of two numbers **a** and **b** could be defined as the **b**-fold application of *s* to **a**:

$$a+b := \underbrace{(s\ \ldots\ (s\ a))\ \ldots}_{b\ \text{times}}$$

In the same way the multiplication of **a** and **b** can be expressed as the **b**-fold application of the "plus" operator, and **a** raised to the power of **b** can be expressed as the **b**-fold application of multiplication:

$$a*b := \underbrace{a + \ldots + a}_{b\ \text{times}} \qquad a^b := \underbrace{a * \ldots * a}_{b\ \text{times}}$$

The game does not end here. The **b**-fold application of the exponentation operator to **a** is called a *power tower* of the height **b**:

$$a\hat{}\hat{}b := \left. a^{\cdot^{\cdot^{a}}} \right\} b\ \text{times}$$

Typical notations for the power tower are *a^^b* and $a^{(4)}b$. The former operator is pronounced "power power" and the latter is pronounced "hyper-4". The *hyper operator* is in fact a generalization of all of the operations described above:

hyper notation	**equivalent form**
$a^{(0)}b$	*(s a)* for any value of *b*
$a^{(1)}b$	$a + b$
$a^{(2)}b$	$a * b$
$a^{(3)}b$	a^b
$a^{(4)}b$	*a ^^ b*

Once again, the implementation of the function looks more innocent than its implications will turn out to be:

```
(require '~nmath)

(define (hyper n a b)
  (cond ((equal n '#0) (+ '#1 a))
        ((equal n '#1) (+ a b))
        ((one b)       a)
        ((equal n '#2) (* a b))
        ((equal n '#3) (expt a b))
        ((equal n '#4) (expt a (hyper n a (- b '#1))))
        ((> n '#4)     (hyper (- n '#1) a (hyper n a (- b '#1))))))
```

Now that we have learned that there is a hyper-4 operator, why should there not be a hyper-5 operator, a tower of power towers? Or a hyper-6 operator or, for that matter, a hyper-100 operator?

We will see.

One thing is certain: no matter which argument of *hyper* is increased, its values grow really fast if *n* is at least 4:

```
(hyper '#4 '#1 '#3) = 1^^1 = 1^1 = 1                              (1 digit)
(hyper '#4 '#2 '#3) = 2^^2 = 2^2 = 2                              (1 digit)
(hyper '#4 '#3 '#3) = 3^^3 = 3^3^3 = 3^27 = 7625597484987       (13 digits)
(hyper '#4 '#4 '#3) = 4^^3 = 4^4^4 = 4^256 = ...               (155 digits)
(hyper '#4 '#5 '#3) = 5^^3 = 5^5^5 = 5^3125 = ...             (2185 digits)
(hyper '#4 '#6 '#3) = 6^^3 = 6^6^6 = 6^46656 = ...           (36306 digits)
```

Iterating the first factor of *hyper* leads to impressive growth. Even $4^{(4)}3$ (or **`(hyper '#4 '#4 '#3)`**) has a result that is *far* larger than the number of atoms in the known universe (which is about 10^{80}).

But the above is only the beginning. As we have seen in the section about the complexity of functions [page 75], iterating the exponent yields *much* faster growth than iterating the base of a power. So:

```
(hyper '#4 '#3 '#1) = 3
(hyper '#4 '#3 '#2) = 3^3 = 27
(hyper '#4 '#3 '#3) = 3^^3 = 3^3^3 = 3^27 = 7625597484987
(hyper '#4 '#3 '#4) = 3^^4 = 3^3^3^3 = 3^3^27 = 3^7625597484987
(hyper '#4 '#3 '#5) = 3^^5 = 3^3^3^3^3 = 3^3^3^27 = 3^3^7625597484987
```

$3^{(4)}4 = 3^{7625597484987}$ is a number with about 3,638,334,640,024 digits. That is 3.6 *trillion* digits. If you print this number using a really small font and squeeze 100,000 digits on a page, you will still need 36 million pages. And $3^{(4)}5$ is 3 raised to the power of *that* number.

But even this growth is shallow compared to iterating the order of the hyper operator itself:

```
(hyper '#1 '#3 '#3) = 3+3 = 6
(hyper '#2 '#3 '#3) = 3*3 = 9
(hyper '#3 '#3 '#3) = 3^3 = 27
(hyper '#4 '#3 '#3) = 3^^3 = 3^3^3 = 3^27 = 7625597484987
(hyper '#5 '#3 '#3) = 3^^^3 = 3^^3^^3 = 3^^7625597484987
```

$3^{(5)}3$ is a power tower of the height 7,625,597,484,987. This is a tower of *7.6 trillion stories*:

$$\left.3^{\cdot^{\cdot^{3}}}\right\} \textit{7,625,597,484,987} \text{ times}$$

Such numbers are *way* beyond human comprehension, and the difference between one result and the next keeps *increasing* exponentially. This is why this kind of growth is called *hyper-exponential* growth.

Do you think that functions with hyper-exponential complexity may have any uses in practice?

Reduce $3^{(6)}3$ to lower-order operations as far as you can get. Attempt to describe the size of $3^{(6)}3$. **(Q15)**

Figure out all sets of arguments of *hyper* for which your computer will finally return a value. Just trying it is obviously not an option, so you will have to do some mental work, but this is what computing science is about.

Do you think that buying a faster computer with more memory would extend the above set?

What does $2^{(100)}2$ evaluate to? . . . . Yes, this is a trick question.

7.6 transposing matrixes

This chapter closes with a useful little one-liner that transposes a matrix. The matrix is stored as a list of rows. The function swaps columns and rows:

```
(transpose '(#abc #def)) => '(#ad #be #cf)
(transpose '(#ad #be #cf)) => '(#abc #def)
```

Here is the code:

```
(define (transpose x) (apply map list x))
```

Can you explain how it works?

8. data structures

8.1 generators

A generator is a data structure that generates a series of values. However, the concept of a generator seems to imply some mutable state, as the following example illustrates by means of a generator that creates the natural numbers:

```
(g) => '#1
(g) => '#2
(g) => '#3
...
```

Each time **g** is called it returns a different value, so **g** cannot be a function in the strict sense. Yet it is possible to implement generators in zenlisp in a purely functional way. This is how it works:

```
(define (generator start step)
  (lambda ()
    (cons start
          (generator (step start) step))))

(define (value g) (car g))
(define (next g) ((cdr g)))
```

The *generator* function is a higher order function that returns a generator. When called, the generator delivers a data structure consisting of the initial value *start* and another generator carrying **`(step start)`** as its value.

The structure is indefinitely recursive, but nevertheless finite. This is because no generator is reduced until requested. For this reason a generator is also called a *lazy* structure. Here is *generator* in action:

```
(load ~nmath)
(generator '#1 (lambda (x) (+ '#1 x)))
          => {closure ()}
     (**) => '(#1 . (closure ()))
(next **) => '(#2 . (closure ()))
(next **) => '(#3 . (closure ()))
...
```

The **`**`** operator always contains the most recent toplevel result, so **`(next **)`** picks up the previous result and runs its embedded generator. Of course, **`**`** itself is stateful and works only in interactive computations, so here is the same example without it:

```
(let ((g ((generator '#1 (lambda (x) (+ '#1 x))))))
  (let ((x (value g))
        (g (next g)))
    (let ((y (value g))
          (g (next g)))
```

```
      (let ((z (value g))
            (g (next g)))
        (list x y z)))))
=> '(#1 #2 #3)
```

Each application of the form **(next g)** appears to have a side effect, because it delivers a new value, but what it really does is to map the current generator to a new one, so it is in fact an ordinary function.

Here is what happens "under the hood":

```
(define (inc x) (+ '#1 x))
(generator '#1 inc)
          =>        (lambda () (cons '#1 (generator (inc '#1) inc)))
     (**) => '(#1 . (lambda () (cons '#2 (generator (inc '#2) inc))))
(next **) => '(#2 . (lambda () (cons '#3 (generator (inc '#3) inc))))
(next **) => '(#3 . (lambda () (cons '#4 (generator (inc '#4) inc))))
```

Can you create a generator that produces the tails of a list? What happens when the end of the list is reached? Can you improve the *generator* function in such a way that it handles such situations more gracefully? **(Q16)**

How are generators related to lists? What are their differences and what do they have in common?

8.2 streams

Basically streams are refined generators. Here is the implementation of the *stream* function which implements the *stream* data type:

```
(define (stream v first filter rest lim final)
  (letrec
    ((find
       (lambda (x)
         (cond ((lim x) x)
               ((filter (first x)) x)
               (t (find (rest x))))))
      (make-stream
        (lambda (v)
          (lambda ()
            (let ((nf (find v)))
              (cond ((lim nf) final)
                    (t (cons (first nf)
                             (make-stream (rest nf))))))))))
    ((make-stream v))))
```

The *v* variable of *stream* has the the same function as the *start* variable of *generator* [page 94] and *rest* is equivalent to *step*. The *make-stream* subfunction is basically equal to *generator*, with some features added.

Using *stream* a generator that produces natural numbers can be created this way:

```
(stream '#1
         id
         (lambda (x) :t)
         (lambda (x) (+ '#1 x))
         (lambda (x) :f)
         :f)
```

The *first* variable is bound to a function that preprocesses each value of the stream before returning it. Because there is nothing to do in this example, the identity function is passed to *stream*.

The *filter* variable binds to a predicate that must reduce to truth for each member of the stream that is to be generated. The above filter just passes through all members.

Lim binds to a predicate checking for the end of the stream. The above predicate returns constant falsity, so the stream is (potentially) infinite. The value of *final* is to be returned when **`(lim x)`** returns truth for some **x**.

Predicates returning constant truth and falsity are common in streams, so they are defined as follows:

```
(define (all x) :t)
(define (none x) :f)
```

The *next* and *value* functions are the same as in the *generator* code:

```
(define (value s) (car s))
(define (next s) ((cdr s)))
```

Using these abbreviations, the above stream of natural numbers can be created in a more comprehensible way:

```
(stream '#1 id all (lambda (x) (+ '#1 x)) none :f)
```

This definition returns a stream ‘‘starting at *one*, returning the *identity* of *all* members, *incrementing members by one*, and having *no* limit’’. The ‘‘final’’ value does not matter in this case because the limit is *none*.

Note that (unlike *generator*) stream functions return streams immediately and not functions returning streams:

```
(stream '#1 id all (lambda (x) (+ '#1 x)) none :f)
         => '(#1 . {closure ()})
(next **) => '(#2 . {closure ()})
(next **) => '(#3 . {closure ()})
...
```

A stream delivering the members of a list would be created as follows:

```
(stream '#abc car all cdr null :f)
         => '(a . {closure ()})
(next **) => '(b . {closure ()})
(next **) => '(c . {closure ()})
(next **) => :f
```

In fact, this expression is so useful that we will give it a name:

```
(define (list->stream v)
  (stream v car all cdr null :f))
```

Stream->list is the reverse operation of *list->stream.* It collects the members of a stream and places them in a list.

```
(define (stream->list s)
  (letrec
    ((s->l
       (lambda (s lst)
         (cond (s (s->l (next s)
                        (cons (value s) lst)))
               (t (reverse lst))))))
    (s->l s ())))
```

BTW, why is it not a good idea to convert a stream of natural numbers — like the one defined above — to a list?

Here follow some higher order functions that can be applied to streams. Most of these functions have exact counterparts in the list domain.

The *stream-member* function locates the first member satisfying the predicate *p* in the given stream. When no such member is found, the default value *d* is returned instead:

```
(define s (list->stream '(#a b #c d)))
(stream-member atom s :f) => '(b . {closure ()})
(stream-member null s :f) => :f
```

Here is the code of *stream-member*. Note that it uses *d* for both detecting the end of the stream and indicating the end of the stream:

```
(define (stream-member p s d)
  (cond ((eq s d) d)
        ((p (value s)) s)
        (t (stream-member p (next s) d))))
```

The *pass* function is another convenience function. It is used to indicate that an embedded stream should be considered to be exhausted when it returns **:f**. However, this is merely a convention used here. Any other value could be used to indicate the end of a stream.

```
(define pass not)
```

Map-stream is like **map**, but works on streams:

```
(require ~nmath)
(map-stream (lambda (x) (* x x))
            (stream '#1 id all (lambda (x) (+ '#1 x)) none :f))
          => '(#1 . {closure ()})
(next **) => '(#4 . {closure ()})
(next **) => '(#9 . {closure ()})
(next **) => '(#16 . {closure ()})
...
```

The *map-stream* function can be implemented using a single invocation of *stream*:

```
(define (map-stream f s)
  (stream s (lambda (s) (f (value s))) all next pass :f))
```

The stream created by it ‘‘starts at the value of *s*, returns *f* applied to *all* members, fetches *next* members from the original stream, *passes* end-of-stream detection to the original stream and returns **:f** as its final value.

So *map-stream* basically constructs a ‘‘stream around a stream’’. The outer stream fetches values from the inner stream and applies a function to each value before returning it. In the above example, the inner stream still creates natural numbers while the outer stream — which is created by *map-stream* — generates squares.

The *filter-stream* function applies a filter to a stream, so that only members with specific properties are generated. It may be considered some kind of ‘‘internal *stream-member*’’ function. Indeed it implemented internally by the *find* function of *stream*. Again, the code is trivial:

```
(define (filter-stream p s)
  (stream s value p next pass :f))
```

Here is how it works:

```
(filter-stream atom (list->stream '(a #b c #d e)))
          => '(a . {closure ()})
(next **) => '(c . {closure ()})
(next **) => '(e . {closure ()})
(next **) => :f
```

Of course, stream functions can be combined:

```
(require ~nmath)
(map-stream (lambda (x) (* x x))
            (stream '#1 id all (lambda (x) (+ '#1 x)) none :f))
          => '(#1 . {closure ()})
(next **) => '(#4 . {closure ()})
(filter-stream even **)
          => '(#4 . {closure ()})
          => '(#16 . {closure ()})
(next **) => '(#36 . {closure ()})
(filter-stream (lambda (x) (zero (remainder x '#7))) **)
          => '(#196 . {closure ()})
(next **) => '(#784 . {closure ()})
(next **) => '(#1764 . {closure ()})
...
```

In this example *map-stream* once again creates a stream of square numbers. Then *filter-stream* filters all even numbers from that stream. Finally, another filter extracts all numbers that are divisible by 7 from the resulting stream. So the final stream generates even square numbers that are divisible by 7.

The last function introduced here appends two streams by wrapping the first stream around the second. When the first stream is exhausted, it returns the second one as its final value:

```
(define (append-streams s1 s2)
  (stream s1 value all next pass s2))
```

Here is *append-streams* in action:

```
(stream->list
  (append-streams (list->stream '#hello-)
                  (list->stream '#world!)))
=> '#hello-world!
```

Can you write an *append-streams** function which is like *append-streams*, but appends a variable number of streams? What should **`(append-streams*)`** reduce to? **(Q17)**

Some of the stream functions can be applied to infinite streams safely and some can not. *Stream->list* is a function that accepts only finite streams:

```
(stream->list (stream 'foo id all id none :f)) => bottom
```

Which of the functions presented above are safe for infinite streams? **(Q18)**

Invent some stream functions of your own. Are there any list operations that cannot be expressed using streams?

8.3 ml-style records

A *record* is a set of ordered tuples similar to an association list. The difference between a record and an alist is that a record has a fixed number of members. Adding or removing members changes the *type* of the record. The following structure resembles a record:

`((`*`food ginger`*`) (`*`type root`*`) (`*`vegetarian :t`*`))`

Each sublist of the record is called a *field*. The car part of each field contains the *tag* of this field and its cadr part contains a value associated with that tag.

The *ML* language [14] provides a highly flexible mechanism for creating and manipulating records, which will be emulated by the following code as far as this is possible in a dynamically typed, purely functional environment.

Records make use of numbers, so any of the math packages has to be loaded. When none of them has been loaded before, **`rmath`** is loaded by default. This solution allows to use records in combination with any of the math packages:

14 ML ("Meta Language") is a statically typed functional programming language invented by Robin Milner et al. at the University of Edinburgh in 1973. It was probably the first language to employ "type inference". See also: `http://www.smlnj.org`.

```
(or (defined 'nmath)
    (defined 'imath)
    (defined 'rmath)
    (load ~rmath))
```

In order to distinguish records from other data types, a unique instance [page 55] of the datum `'(%record)` is created. This datum will be used to tag records.

```
(define record-tag (list '%record))
```

The below procedures determine the types of given forms. Note that both closures and records are lists at the same time. There is no way to implement more strict type checking in zenlisp.

```
(define (pair-p x) (not (atom x)))

(define (boolean-p x)
  (or (eq x :t)
      (eq x :f)))

(define (closure-p x)
  (and (pair-p x)
       (eq (car x) 'closure)))

(define (record-p x)
  (and (pair-p x)
       (eq (car x) record-tag)))
```

List->record turns a list of two-element lists (tag/value tuples) into a record:

```
(list->record '((food marmelade) (type processed)))
  => ((%record) (food marmelade) (type processed))
(list->record '(foo bar))
  => bottom
```

It also checks whether the pairs have the appropriate format, but it does not check for duplicate tags. Feel free to improve the code on your own.

```
(define (list->record a)
  (letrec
    ((valid-fields-p
       (lambda (a)
         (or (null a)
             (and (pair-p (car a))
                  (atom (caar a))
                  (pair-p (cdar a))
                  (null (cddar a))
                  (valid-fields-p (cdr a)))))))
    (cond ((valid-fields-p a) (cons record-tag a))
          (t (bottom 'bad-record-structure a)))))
```

The *record* function is the principal record constructor. It assembles a record from a set of tag/value tuples:

```
(record '(foo bar) '(baz goo)) => ((%record) (foo bar) (baz goo))
```

It is based on *list->record*, but evaluates its individual arguments before constructing the record:

```
(define (record . x) (list->record x))
```

Note that you *cannot* create record literals like `'((%record) (foo bar))`, because the `(%record)` part of that structure is not identical to the unique instance bound to *record-tag*:

```
(record-p (record '(foo bar)))     => :t
(record-p '((%record) (foo bar))) => :f
```

Record->list is the reverse operation of *list->record*.

```
(define (record->list r)
  (cond ((record-p r) (cdr r))
        (t (bottom 'expected-record-got r))))
```

The *record-field* function extracts the field with a given tag from a record and *record-ref* extracts the value associated with a given tag:

```
(record-field (record '(a #1) '(b #2)) 'b) => '(b #2)
(record-ref   (record '(a #1) '(b #2)) 'b) => #2
```

Both of them reduce to bottom when either their first argument is not a record or the second argument does not occur as a tag in the given record.

```
(define (record-field r tag)
  (let ((v (assq tag (record->list r))))
    (cond (v v)
          (t (bottom 'no-such-tag
                     (list 'record: r 'tag: tag))))))

(define (record-ref r tag) (cadr (record-field r tag)))
```

Type-of returns a symbol that indicates the type of a given form. Note that the list does not occur in *type-of*. It returns `'pair` for pairs (and hence also for lists) and even for empty lists.

```
(define (type-of x)
  (cond ((boolean-p x)  'boolean)
        ((null x)       'pair)
        ((atom x)       'symbol)
        ((number-p x)   'number)
        ((record-p x)   'record)
        ((closure-p x)  'function)
        ((pair-p x)     'pair)
        (t (bottom 'unknown-type x))))
```

Two records are *equal*, if

– they have the same number of fields;
– their fields have the same tags;
– fields with identical tags have equal values.

The fields of two equal records need not appear in the same order. For example, the following two records are equal:

```
(record '(foo #1) '(bar #2))
(record '(bar #2) '(foo #1))
```

If records contain records, embedded records are compared recursively. The *record-equal* predicate compares records:

```
(define (record-equal r1 r2)
  (letrec
    ((equal-fields-p
       (lambda (r1 r2)
         (cond ((null r1) :t)
               (t (let ((x (assq (caar r1) r2)))
                    (and x
                         (equal (cadar r1) (cadr x))
                         (equal-fields-p (cdr r1) r2))))))))
    (let ((lr1 (record->list r1))
          (lr2 (record->list r2)))
      (and (= (length lr1) (length lr2))
           (equal-fields-p lr1 lr2)))))
```

The **`equal`** predicate is extended in order to handle records, too:

```
(define (equal a b)
  (cond ((eq a b) :t)
        ((and (pair-p a) (pair-p b))
          (and (equal (car a) (car b))
               (equal (cdr a) (cdr b))))
        ((record-p a)
          (and (record-p b)
               (record-equal a b)))
        (t :f)))
```

The *signature* of a record **r** is another record that contains the same tags, but instead of the values of **r** it contains the *types* of its values, e.g.:

```
(record-signature (record '(food apple) '(weight #550) '(vegetarian :t)))
=> '((%record) (food symbol) (weight number) (vegetarian boolean))
```

A record containing embedded records has a recursive signature:

```
(record-signature (record (list 'p1 (record '(x #0) '(y #0)))
                          (list 'p2 (record '(dx #0) '(dy #0)))))
=> '((%record) (p1 (record ((%record) (x number) (y number))))
               (p2 (record ((%record) (dx number) (dy number)))))
```

The *record-signature* function creates the signature of a record:

```
(define (record-signature r)
  (letrec
    ((make-sig
       (lambda (x)
         (map (lambda (x)
                (cond ((record-p (cadr x))
                        (list (car x)
                              (list (type-of (cadr x))
                                    (record-signature (cadr x)))))
                      (t (list (car x) (type-of (cadr x))))))
              x))))
    (list->record (make-sig (record->list r)))))
```

The *record-set* function creates a fresh record in which the value of the given field is replaced with a new value:

```
(define r (record '(food cucumber)))
                          r => '((%record) (food cucumber))

(record-set r 'food 'melon) => '((%record) (food melon))
                          r => '((%record) (food cucumber))
```

Note that *record-set* does not alter the original record. When the given tag does not occur in the given record or the value associated with the tag has a different type than the new value, *record-set* yields bottom:

```
(record-set (record '(food salt)) 'zzz 'baz) => bottom ; unknown tag

(record-set (record '(food salt)) 'food :f)  => bottom ; type mismatch
```

When replacing values of the type record, the old and the new value must have the same signature:

```
(define r (record (list 'menu (record '(food apple)))))

(record-set r 'food (record '(food (x y z))))
=> bottom ; type mismatch, expected: ((%record) (food symbol))
                                got: ((%record) (food pair))

(record-set r 'menu (record '(food orange)))
=> '((%record) (menu ((%record) (food orange))))
```

Here is the code of the *record-set* function.

```
(define (record-set r tag v)
  (letrec
    ((subst
       (lambda (r old new)
         (cond ((null r) ())
               ((eq old (car r))
                 (cons new (cdr r)))
               (t (cons (car r)
                        (subst (cdr r) old new))))))
```

```
     (type-mismatch
       (lambda ()
         (bottom 'type-mismatch
                 (list 'record: r 'tag: tag 'value: v)))))
    (let ((f (record-field r tag)))
      (let ((b (cdr f)))
        (cond ((eq (type-of (car b)) (type-of v))
                (cond ((or (not (record-p v))
                           (record-equal
                             (record-signature (car b))
                             (record-signature v)))
                        (subst r f (list (car f) v)))
                      (t (type-mismatch))))
                (t (type-mismatch)))))))
```

Because *record-set* only replaces values of matching types, it makes records as type-safe as ML records. Note that there is no need to declare record types in order to achieve type safety. The type of a record is determined by extracting its signature.

Explicit type checking or dispatch can be added to a function by means of the *assert-record-type* and *record-type-matches-p* functions.

Record-type-matches-p is a predicate that returns truth if a given record matches a given signature. It is merely an abbreviation:

```
(define (record-type-matches-p sig r)
  (record-equal sig (record-signature r)))
```

It is used as follows:

```
(define point-type (record-signature (record '(x #0) '(y #0))))
...
(define (some-function x)
  (cond ((record-type-matches-p point-type x)
          code handling point records...)
        (t code handling other types ...)))
```

Assert-record-type is similar, but used to *make sure* that an argument has a given type:

```
(define (some-function r)
  (function-expecting-a-point (assert-record-type point-type r)))
```

As long as a record passed to *assert-record-type* has the expected signature, the function simply returns the record. When its type does not match, its aborts the computation and reduces to bottom. Here is the code of *assert-record-type*:

```
(define (assert-record-type sig r)
  (cond ((not (record-type-matches-p sig r))
          (bottom 'record-type-assertion-failed
                  (list 'signature: sig 'record: r)))
        (t r)))
```

What signature does the signature of a record have? And the signature of the signature of a record? Give some examples. **(Q19)**

Of course, the record type is most useful in languages that support mutation of values, so this implementation is merely a proof of concept. Can you imagine any real use for the code of this section?

Which modification(s) would you apply to the code when porting it to a language supporting mutable values (like Scheme)?

9. compilers

9.1 translating infix to prefix

The *infix->prefix* function presented in this section implements a so-called *recursive descent parser*. A *parser* is a program that analyses the syntactical structure of its input and transforms it into some other representation. In the example presented in this section, the ‘‘other representation’’ is that of zenlisp expressions. *Recursive descent* is a parsing technique. It will be explained in detail in this section.

The first question when designing a parser is how to represent its input. In the real world, this would most probably be a string or a ‘‘text file’’, but because zenlisp does not provide either of them, lists will be used instead. For example, the formula

$x^2 + y$

would be written as

```
'#x^2+y
```

The *infix->prefix* program will analyze formulae of the above form and translate them to corresponding zenlisp forms, e.g.:

```
(infix->prefix '#x^2+y)      => '(+ (expt x '#2) y)
(infix->prefix '#x*2+y*3)    => '(+ (* x '#2) (* y '#3))
(infix->prefix '#x*[2+y]*3) => '(* (* x (+ '#2 y)) '#3)
```

The parser will recognize the following symbols in its input:

Input	**Meaning**
`[a-z]`	symbol (single letters only)
`[0-9]+`	integer numbers (sequences of digits)
`+`	addition
`-`	subtraction or negation (depends on context)
`*`	multiplication
`/`	division
`^`	exponentation
`[]`	grouping of subexpressions

In order to describe the input of a parser in detail, though, a little digression is in order.

9.1.1 formal grammars

Technically speaking, the input of the *infix->prefix* program will be a *formal language*. The symbols given at the end of the previous section are the *lexemes* of that language. Any non-empty sequence of such lexemes is a *sentence* of that language (although not necessarily a *well-formed* one). The following sequences are sentences:

```
aaaa
a+b
a+-b^
[x]y
]]]
```

Just like natural languages formal languages have grammars that are used to construct *well-formed sentences*. A well-formed sentence of a formal language is also called a *program* of that language. Intuition may tell you that the following sentences are programs:

```
a+b
x*y+z
x-17
```

But what about these:

```
xyz
a--b
p[q]
```

Intuition is fine but hard to implement in a parser, so we need some means of describing a grammar *formally*. This is where *BNF* (“Backus Normal Form” or “Backus Naur Form”) comes into play. BNF is a notation for describing the grammars of programming languages formally. The basic building stone of BNF descriptions is the *production*. Productions look like this:

```
<sum> := symbol '+' symbol
       | symbol '-' symbol
```

The “`:=`” operator reads “is defined as” or “can be written as”. The “`|`” denotes a logical or. So the above production says: “a `<sum>` can be written as a `symbol` followed by + and another `symbol` or as a `symbol` followed by ‘−’ and another `symbol`”.

Each name that stands alone denotes a lexeme, which is also called a *terminal symbol* in compiler speak. Names like `symbol` typically represent classes of symbols. In the language we are about to define, `symbol` would denote the class containing the lexemes

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

and `number` would represent the (infinite) class containing the lexemes

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ...
```

A name that is enclosed in apostrophes is a terminal symbol that represents itself, so `'+'` represents the lexeme + and `'['` represents the lexeme `[`.

Names enclosed in angle brackets, like `<sum>`, are so-called *non-terminal symbols*. They represent productions. The conventions used for terminals, non-terminals and even the operators differ between textbooks, but the fundamental principles are always the same: the lefthand side of a production gives a name to the production and the righthand side describes what the lefthand side can be replaced with. For example, according to the above production, these are `<sum>`s (here the generic arrow means ''according to the rule''):

```
a+b  -->  symbol '+' symbol
x-y  -->  symbol '-' symbol
```

Any sentence not matching the rules of `<sum>` is not a valid program of `<sum>`. The term *rule* is sometimes used as a synonym of ''production''. In this text, though, ''rule'' will be used to refer to one alternative of a production, so the `<sum>` production has two rules. These rules could also be written as separate productions:

```
<sum> := symbol '+' symbol
<sum> := symbol '-' symbol
```

This is rarely done, though, because the ''or'' operator makes productions more readable.

The omnipresent principle of recursion also plays a central role in BNF productions. It is used to describe sentences of variable length:

```
<a*> := 'a'
      |  'a' <a*>
```

The production `<a*>` matches any positive number of `a`s. Its rules say that `<a*>` may be replaced by a single `a` or by an `a` followed by another `<a*>` (which in turn may be either a single `a` or an `a` followed by another `<a*>` (which ... you get the idea)), so these rules ''produce'' [15] the sentences

```
a       -->  'a'
aa      -->  'a' <a*> 'a'
aaa     -->  'a' <a*> 'a' <a*> 'a'
aaaa    -->  'a' <a*> 'a' <a*> 'a' <a*> 'a'
aaaaa   -->  'a' <a*> 'a' <a*> 'a' <a*> 'a' <a*> 'a'
...
```

To say that a ''production produces'' a set of sentences implies that the production matches these sentences. The set of sentences produced by a production is exactly the set of programs accepted by that production.

The principle of recursion can be used, for instance, to form `<sum>`'s with any number of operands:

```
<sum> := symbol
       |  symbol '+' <sum>
       |  symbol '-' <sum>
```

15 Yes, this is why a set of rules is called a ''production''.

Here are some of the sentences produced by this version of `<sum>`:

```
x       —>  symbol
x+y     —>  symbol '+' <sum> symbol
x+y-z   —>  symbol '+' <sum> symbol '-' <sum> symbol
```

The above production can (almost) be used to describe a part of the language accepted by the *infix->prefix* parser. However, the parser will accept not just symbols as operands but also numbers, and it will accept terms and exponents, too. One production is not enough to cover all of these, so multiple productions will be combined to form a *grammar*. The following grammar accepts a language with both numbers and symbols as operands in sums:

```
<sum> := <factor>
       |  <factor> '+' <sum>
       |  <factor> '-' <sum>

<factor> := symbol
          |  number
```

In real-world math formulae operations like multiplication and division "bind stronger" than, for example, addition and subtraction. A compiler writer would say that multiplication and division have a "higher precedence" than addition and subtraction. Precedence is easy to implement in grammars (see the above grammar for the definition of `<factor>`):

```
<term> := <factor>
        |  <factor> '*' <term>

<sum> := <term>
       |  <term> '+' <sum>
```

Here `<term>` works in the same way as `<sum>` in the previous grammar. `<Sum>`s are now composed of `<term>`s. Because a complete `<term>` has to be parsed before a `<sum>` can be produced, `<term>`s bind stronger than `<sum>`s. In other words: a `<sum>` is a `<term>` followed by optional `<sum>` operations.

Let us check some intuitively well-formed sentences against this grammar:

```
x       —>  <sum> <term> <factor> 'x'
x+y     —>  <sum> <term> <factor> 'x' '+' <sum> <term> <factor> 'y'
x+y*z   —>  <sum> <term> <factor> 'x' '+' <sum> <term> <factor> 'y'
             '*' <term> <factor> 'z'
x*y+z   —>  <sum> <term> <factor> 'x' '*' <term> <factor> 'y'
             '+' <sum> <term> <factor> 'z'
```

Okay, this is the point where things become a bit messy, because the linear representation is not really suitable for representing sentences. This is why the output of parsers is typically presented in tree form.

The tree in figure 5 shows the *syntax tree* of the formula `x+y*z`. A syntax tree is sometimes also called a *parse tree*. The square boxes represent non-terminals, the circles terminals. Following the outer edge of the tree visits the terminals in the order in which they appear in the original formula.

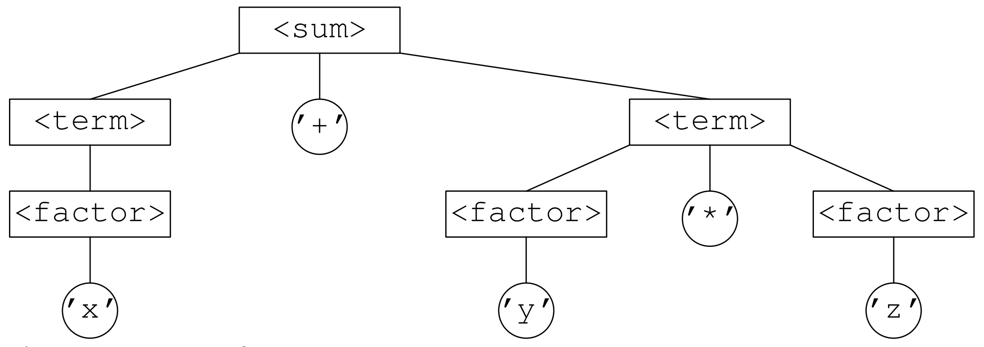

Fig. 5 – syntax tree of `x+y*z`

Because a `<term>` can be part of a `<sum>`, but a `<sum>` can never be part of a `<term>`, `<term>`s are always contained in `<sum>` trees and therefore, term operations are visited before sum operations when traversing the tree using "depth-first" traversal. As a consequence of this order, term operations have a higher precedence than sum operations.

Depth-first traversal of a tree means that subtrees are always visited before processing parents. For instance, a tree can be converted to *reverse polish notation* (*suffix notation*) by visiting the left branch of each non-terminal node first, then visiting the right branch and finally emitting the terminal attached to the node (if any). Traversing the above tree would yield:

```
x y z * +
```

Emitting the operand *before* descending into branches would yield prefix notation:

```
+ x * y z
```

Adding parentheses gives a zenlisp program:

```
(+ x (* y z))
```

Note that the precedence of the sum and term operators is preserved in all notations: tree, suffix, and prefix.

We now know how to define grammars, represent parsed text, and even how to generate zenlisp programs. What is missing is the full grammar of the input language. Here it is:

```
<sum> := <term>
       |  <term> '+' <sum>
       |  <term> '-' <sum>

<term> := <power>
        |  <power> '*' <term>
        |  <power> <term>
        |  <power> '/' <term>
```

```
<power> := <factor>
        |  <factor> '^' <power>

<factor> := symbol
         |  number
         |  '-' <factor>
         |  '[' <sum> ']'
```

Each well-formed sentence accepted by the *infix-prefix* parser is a production of `<sum>`. Note that the rule

```
<term> := <power> <term>
```

allows to abbreviate `x*y` as `xy`. Of course, this is only possible because variables are single-letter lexemes. The rule

```
<factor> := '[' <sum> ']'
```

gives a `<sum>` (no matter which operators it eventually contains) the precedence of a `<factor>`, thereby allowing to group subexpressions just like in math formulae. Also note that the minus prefix (which negates a factor) has the highest possible precedence, so

```
-x^2
```

actually means

```
(-x)^2
```

Now that the grammar for the *infix->prefix* parser has been specified formally, there is no need to rely on intuition any longer, and the implementation of the parser is quite straight-forward.

Well, almost straight-forward. There is one subtle detail left to discuss.

9.1.2 left versus right recursion

The recursive productions used in the grammars shown so far are so-called *right-recursive* productions. They are termed so because the recursive rules of each production recurse at the rightmost end, like this:

```
<diff> := <factor>
       |  <factor> '-' <diff>
```

A right-recursive parser for this small grammar can be easily packaged in a function (although some of the details will be explained later and are left to the imagination of the reader for now):

```
(define (diff x)
  (let ((left (factor x)))
    (cond ((null (rest left)) left)
          ((eq (car-of-rest left) '-)
            (let ((right (diff (cdr-of-rest left))))
              (list '- (expr left) (expr right))))
          (t left))))
```

This parser recognizes chains of "–" operators just fine, but the parse tree generated by it associates operators to the right, as shown in figure 6.

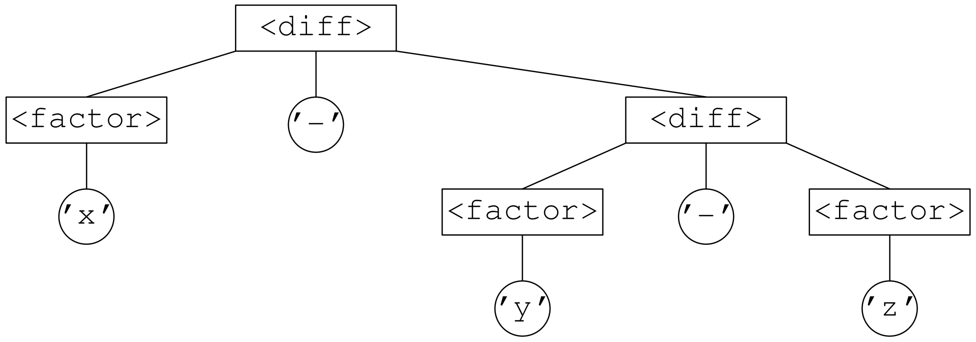

Fig. 6 – right-associative syntax tree of `x-y-z`

When traversing this tree, the resulting prefix expression would be

```
(- x (- y z))
```

which in turn translates back to

x-(y-z)

It clearly *should* give *(x-y)-z*, though. In other words, the parser gives the "–" operator the wrong *associativity*:

```
x-y-z  =  (- x (- y z))  ; right-associative
x-y-z  =  (- (- x y) z)  ; left-associative
```

Left-associativity is needed, but right-associativity is delivered. In a grammar this is easy to fix by simply rewriting it in a *left-recursive* way, where recursion occurs at the beginning of each rule:

```
<diff> := <factor>
       |  <diff> '-' <factor>
```

Unfortunately, this approach cannot be implemented in the way outlined above, as can be seen in the following code fragment:

```
(define (diff x)
  (let ((left (diff x)))
    (cond ((null (rest left)) left)
          ...)))
```

Because this hypothetical parser function would recurse immediately, it would never reach its trivial case and hence recurse indefinitely.

By rewriting the function slightly, left recursion can be eliminated, though:

```
(define (diff2 out in)
  (cond ((null in)
          out)
        ((eq (car in) '-)
          (let ((right (factor (cdr in))))
            (diff2 (list '- out (expr right))
                   (rest right))))
        (t out)))

(define (diff x)
  (let ((left (factor x)))
    (diff2 (expr left) (rest left))))
```

This version of the *diff* function passes the first factor to *diff2* which then collects "−" operators (if any). *Diff2* descends immediately into *factor* rather than into itself and *then* recurses to collect more factors. To prove that it actually builds a left-associative expressions is left as an exercise to the reader.

9.1.2 implementation

The parser implemented in this section is a so-called *recursive descent parser*. It implements each production of a formal grammar in a separate function (e.g. the *factor* function implements the rules of `<factor>`, etc). Like productions of a grammar, these functions form a hierarchy. Lexemes are matched by *decending* into this hierarchy until a function is found that recognizes the given lexeme. The method is called *recursive* descent, because functions recurse to accept variable-length input or to ascend back to higher levels of the hierarchy (e.g. for handling parenthesized subexpressions).

Here comes the code:

```
(require '~rmath)

(define (infix->prefix x)
  (letrec
```

Check whether *x* is a symbol:

```
    ((symbol-p
       (lambda (x)
         (and (memq x '#abcdefghijklmnopqrstuvwxyz) :t)))
```

Collect a numeric literal from the input and return a list containing the literal and the rest of the input, e.g.:

```
(number '#123+x ()) => '('#123 #+x)
```

(The result really does contain a quoted number, this is not a mistake.)

```
    (number
      (lambda (x r)
        (cond ((or (null x)
                   (not (digitp (car x))))
                (list (list 'quote (reverse r)) x))
              (t (number (cdr x) (cons (car x) r))))))
```

Extract a symbol from the input (analogous to *number*).

```
    (symbol
      (lambda (x)
        (list (car x) (cdr x))))
```

The following convenience functions are used to access the individual parts of partially translated formulae. For example, the *number* function may return the value `'('#1 #-x*7)`. The car part of such a result is a zenlisp expression, which is extracted using *expr*:

```
(expr '('#1 #-x*7)) => '#1
```

Rest extracts the remaining input that still is to be parsed, *car-of-rest* the first character of the rest, and *cdr-of-rest* the rest with its first character removed:

```
       (rest '('#1 #-x*7)) => '#-x*7
(car-of-rest '('#1 #-x*7)) => '-
(cdr-of-rest '('#1 #-x*7)) => '#x*7
```

While parsing a formula, the intermediate steps are always kept in expr/rest tuples.

```
    (expr car)
    (rest cadr)
    (car-of-rest caadr)
    (cdr-of-rest cdadr)
```

The *factor* function parses a factor as described by the `<factor>` production. Like all parsing functions, it returns an expr/rest tuple:

```
(factor '#-123+456) => '((- '#123) '#+456)
```

Factor, being at the bottom of the recursive descent chain, cannot accept empty input. Hence it aborts with a "syntax error" when an empty list is passed to it. It also reports a syntax error when the lexeme at the beginning of its input does not match any of its rules, e.g.:

```
(factor '#+++) => bottom
```

When it finds an opening parenthesis, it makes sure that a closing parenthesis follows after the `<sum>` between the parentheses.

```
    ; <factor> := '[' <sum> ']'
    ;          |  '-' <factor>
    ;          |  number
    ;          |  symbol
    (factor
      (lambda (x)
        (cond ((null x)
                (bottom 'syntax 'error 'at: x))
              ((eq (car x) '[)
                (let ((xsub (sum (cdr x))))
                  (cond ((null (rest xsub))
                          (bottom 'missing-right-paren))
                        ((eq (car-of-rest xsub) '])
                          (list (expr xsub) (cdr-of-rest xsub)))
                        (t (bottom 'missing-right-paren)))))
              ((eq (car x) '-)
                (let ((fac (factor (cdr x))))
                  (list (list '- (expr fac)) (rest fac))))
              ((digitp (car x))
                (number x ()))
              ((symbol-p (car x))
                (symbol x))
              (t (bottom 'syntax 'error 'at: x)))))
```

Power implements the `<power>` production. It always parses one factor (which is why it does not accept empty input) and then recurses to collect more factors connected by ^ operators to the expression parsed so far:

```
(power '#x^y^z+5) => '((expt x (expt y z)) #+5)
```

It stops parsing when a factor is followed by something that is not a ^ operator (or when the input is exhausted).

Note that right recursion is desired in *power* because powers actually do associate to the right.

```
    ; <power> := <factor>
    ;          |  <factor> ^ <power>
    (power (lambda (x)
      (let ((left (factor x)))
        (cond ((null (rest left)) left)
              ((eq (car-of-rest left) '^)
                (let ((right (power (cdr-of-rest left))))
                  (list (list 'expt (expr left) (expr right))
                        (rest right))))
              (t left)))))
```

Term is similar to *power* but applies the left recursion hack described in the previous subsection. It accepts the `*` and `/` operators instead of `^`. It also accepts `xx` as an abbreviation for `x*x`:

```
    ; term := power
    ;       | power Symbol
    ;       | power * term
    ;       | power / term
    (term2
      (lambda (out in)
        (cond ((null in) (list out in))
              ((symbol-p (car in))
                (let ((right (power in)))
                  (term2 (list '* out (expr right))
                         (rest right))))
              ((eq (car in) '*)
                (let ((right (power (cdr in))))
                  (term2 (list '* out (expr right))
                         (rest right))))
              ((eq (car in) '/)
                (let ((right (power (cdr in))))
                  (term2 (list '/ out (expr right))
                         (rest right))))
              (t (list out in)))))
    (term
      (lambda (x)
        (let ((left (power x)))
          (term2 (expr left) (rest left)))))
```

All of the parsing functions follow the same scheme, so `sum` is basically like `factor`, `power`, and `term`. It differs from them only in the operators being accepted. In a later chapter, a more general approach will be shown that avoids this code duplication.

```
    ; sum := term
    ;      | term + sum
    ;      | term - sum
    (sum2
      (lambda (out in)
        (cond ((null in) (list out in))
              ((eq (car in) '+)
                (let ((right (term (cdr in))))
                  (sum2 (list '+ out (expr right))
                        (rest right))))
              ((eq (car in) '-)
                (let ((right (term (cdr in))))
                  (sum2 (list '- out (expr right))
                        (rest right))))
              (t (list out in)))))
    (sum
      (lambda (x)
        (let ((left (term x)))
          (sum2 (expr left) (rest left))))))
```

The body of *infix->prefix* passes its argument to `sum`. When the code in that argument could be parsed successfully, the rest part of the resulting tuple will be empty. Otherwise none of the functions in the recursive descent chain had a rule for handling the lexeme at the beginning of the rest, so a non-empty rest indicates a syntax error. E.g.:

```
(sum 'x+y@z) => '((+ x y) #@z)
```

When the rest part of the tuple is empty, the body simple returns the expression part, which contains the complete prefix expression at this point.

```
  (let ((px (sum x)))
    (cond ((not (null (rest px)))
            (bottom (list 'syntax 'error 'at: (cadr px))))
          (t (expr px))))))
```

The *infix->prefix* program does not explicitly create a syntax tree. Nevertheless it uses the approach described in this section to convert infix to prefix notation. How does it do this? What are the inner nodes of the tree? **(Q20)**

Some people would argue that making $-x^2$ equal to $(-x)^2$ is a bad idea. Can you change the precedence of the unary minus operator in such a way that it still binds stronger than the term operators `*` and `/` but not as strong as exponentiaton (`^`)? Implement your modification as a BNF grammar as well as in zenlisp code. **(Q21)**

9.2 translating prefix to infix

As its name suggests, *prefix->infix* is the inverse function of *infix->prefix*. It takes a prefix expression represented by a zenlisp form and turns it into an infix expression. Like its cousin, it preserves precedence and associativity. It adds parentheses where necessary:

```
(prefix->infix '(+ (expt x '#2) y))       => '#x^2+y
(prefix->infix '(+ (* x '#2) (* y '#3))) => '#x*2+y*3
(prefix->infix '(- (- a b) (- c d)))      => '#a-b-[c-d]
```

A combination of *infix->prefix* and *prefix->infix* can be used to remove (most) superfluous parentheses from a formula:

```
(prefix->infix (infix->prefix '#[a+b]-[c+d])) => '#a+b-[c+d]
```

There does not appear to be a commonly used named for functions like *prefix->infix*. What they do is a mixture of tree traversal and code synthesis for a virtual “infix machine”. The largest part of the program deals with the generation of parentheses.

Here is the code:

```
(define (prefix->infix x)
  (letrec
```

The *ops* alist maps zenlisp functions to infix operators, *left* contains all functions that map to left-associative operators, and *precedence* contains operators in groups of descending precedence.

```
    ((ops '((+ . +) (- . -) (* . *) (/ . /) (expt . ^)))
     (left '#+-*/)
     (precedence '(high ([]) (expt) (* /) (+ -) low))
```

The following predicates are used to check properties of forms. For instance, *function-p* checks whether a form denotes a function and *left-assoc-p* evaluates to truth when its argument is a function resembling a left-associative operator. These functions should be pretty much self-explanatory:

```
(function-p
  (lambda (x)
    (and (memq x '(+ - * / expt)) :t)))
(left-assoc-p
  (lambda (x)
    (and (memq x left))))
(symbol-p
  (lambda (x)
    (and (memq x '#abcdefghijklmnopqrstuvwxyz) :t)))
(numeric-p
  (lambda (x)
    (and (not (atom x))
         (eq (car x) 'quote))))
(atomic-p
  (lambda (x)
    (or (function-p x)
        (symbol-p x)
        (numeric-p x))))
```

Unary-p checks whether a form represents a unary function application:

```
(unary-p
  (lambda (x)
    (and (not (null (cdr x)))
         (null (cddr x)))))
```

The *higher-prec-p* function finds out whether the formula *x* has a higher precedence than the formula *y*. For instance:

```
(higher-prec-p '#1 '(+ a b))       => :t
(higher-prec-p '(* a b) (+ a b))   => :t
(higher-prec-p '(- a) (expt a b)) => :t
(higher-prec-p '(- a) 'a)          => :f
```

Atomic forms (symbols, numbers) have the highest precedence (*because* they are atomic), followed by unary operators and the precedence rules expressed in the *precedence* list.

```
(higher-prec-p
  (lambda (x y)
    (letrec
      ((hpp (lambda (x y prec)
              (cond ((atom prec) :f)
                    ((memq x (car prec))
                      (not (memq y (car prec))))
                    ((memq y (car prec)) :f)
                    (t (hpp x y (cdr prec)))))))
```

```
          (cond ((atomic-p x) (not (atomic-p y)))
                ((atomic-p y) :f)
                ((unary-p x) (not (unary-p y)))
                ((unary-p y) :f)
                (t (hpp (car x) (car y) (cdr precedence)))))))
```

Paren places parentheses (`[` and `]`) around the given expression, but never around atoms.

```
    (paren
      (lambda (x)
        (cond ((atomic-p x) x)
              (t (list '[] x)))))
```

The *add-parens* function adds parenthesis tags to an expression:

```
(add-parens '(* (+ a b) c)) => '(* ([] (+ a b)) c)
```

The `[]` ("parens") symbol indicates that the following subexpression must be put in parentheses to preserve precedence when converting it to infix. *Add-parens* tags only those subexpressions that really need explicit grouping:

```
(add-parens '(+ (* a b) c)) => '(+ (* a b) c)
```

When an atomic form is passed to *add-parens*, it simply returns it, otherwise it first processes subforms by recursing through **map**. Finally it applies the precedence rules we know.

When the current formula is the application of a unary function and the argument is neither atomic nor another unary function, the argument is put in parentheses.

When the current formula is the application of a left-associative binary function, parentheses are placed around the left argument if the operator of the formula has a higher precedence than that of the argument:

```
(add-parens '(* (+ a b) c)) => '(* ([] (+ a b)) c)
```

The second argument is put in parentheses if it has not a higher precedence than the operator of the formula (that is, if its predecence is lower than or equal to the operator of the formula):

```
(add-parens '(* a (+ b c))) => '(* a ([] (+ b c)))
```

The latter rule also tags operations that are grouped to the right at the same precedence level.

```
(add-parens '(- a (- b c))) => '(- a ([] (- b c)))
```

The rules for right-associatve operations (the catch-all clause of the inner **cond** of *add-parens*) are similar to the above, but the rules are changed in such a way that operations that group to the left at the same precedence level are tagged.

```
    (add-parens
      (lambda (x)
        (cond
          ((atomic-p x) x)
          (t (let ((x (map add-parens x)))
               (cond ((unary-p x)
                       (cond ((atomic-p (cadr x)) x)
                             ((unary-p (cadr x)) x)
                             (t (list (car x)
                                      (paren (cadr x))))))
                     ((left-assoc-p (car x))
                       (list (car x)
                             (cond ((higher-prec-p x (cadr x))
                                     (paren (cadr x)))
                                   (t (cadr x)))
                             (cond ((higher-prec-p (caddr x) x)
                                     (caddr x))
                                   (t (paren (caddr x))))))
                     (t (list (car x)
                              (cond ((higher-prec-p (cadr x) x)
                                      (cadr x))
                                    (t (paren (cadr x))))
                              (cond ((higher-prec-p x (caddr x))
                                      (paren (caddr x)))
                                    (t (caddr x)))))))))))
```

The *infix* function traverses the tree represented by a zenlisp form and emits an infix expression. It puts parentheses around subexpressions tagged by `[]`:

```
(infix '(* (+ x y) z))      => '#x+y*z
(infix '(* ([] (+ x y)) z)) => '#[x+y]*z
```

Infix also checks the consistency (syntax) of the zenlisp expression and report errors.

```
     (infix
       (lambda (x)
         (cond
           ((numeric-p x)
             (cadr x))
           ((symbol-p x)
             (list x))
           ((and (eq (car x) '-)
                 (not (atom (cdr x)))
                 (null (cddr x)))
             (append '#- (infix (cadr x))))
           ((and (eq (car x) '[])
                 (not (atom (cdr x)))
                 (null (cddr x)))
             (append '#[ (infix (cadr x)) '#]))
           ((and (not (atom x))
                 (not (atom (cdr x)))
                 (not (atom (cddr x)))
                 (null (cdddr x))
                 (function-p (car x)))
```

```
                (append (infix (cadr x))
                        (list (cdr (assq (car x) ops)))
                        (infix (caddr x))))
            (t (bottom (list 'syntax 'error: x)))))))
```

The main body simply combines *add-parens* and *infix*.

```
    (infix (add-parens x))))
```

Can you rewrite *prefix->infix* in such a way that it puts parentheses around *all* operations, thereby making precedence explicit? What practical applications could such a transformation have? **(Q22)**

Can you make *prefix->infix* emit reverse polish notation (RPN, suffix notation) instead of infix notation? E.g.:

```
(prefix->rpn '(* (+ x y) z)) => '#xy+z*
```

Does RPN ever need parentheses? Why? **(Q23)**

Note that *prefix->infix* sometimes adds superflous parentheses:

```
(prefix->infix '(+ x (+ y z))) => '#x+[y+z]
```

This is because the program does not implement *commutativity*. An operator **o** is commutative, if chains of **o** operations are independent of associativity:

(a **o** *b)* **o** *c* = *a* **o** *(b* **o** *c)*

The `+` and `*` operators are commutative. Can you implement rules in *add-parens* that recognize cummutativity and skip parentheses where possible?

9.3 regular expressions

A *regular expression* (RE) is a pattern that is used to match sequences of characters. An RE is more general than a string, because it allows to include "special" characters which match classes or characters or sequences of characters. The details will be explained immediately.

The functions introduced in this chapter implement what is now called "basic" regular expressions. This is the RE format used in "traditional" Unix and early versions of the `grep(1)` utility.

A regular expression consists of a set of characters and operators. Most characters simply match themselves, so the RE `foo` would match the sequence `foo`, or even the sequence `afoob` because it contains `foo`.

The following characters have special meanings in REs: [16]

16 Unix-style REs would use the dot (`.`) instead of the underscore (`_`), but this cannot be done in zenlisp because the dot is reserved for dotted pairs.

```
_ [ ] ^ $ * + ? \
```

When one of these characters is to be matched literally, it must be prefixed with a backslash (\). Otherwise it is interpreted as an operator. The meanings of the operators are as follows:

`[c1...]`	A *character class* matches any character contained in the square brackets.
`[^c1...]`	When the first character between the brackets is ^, the class matches any character *not* contained in it.
`[c1-c2 ...]`	When a minus sign occurs in a class, it is replaced with the characters that occur between the character in front of the minus sign and the character following the sign, e.g.: `[0-9]` expands to `[0123456789]`.
`_`	This is the class containing *all* characters, so it matches any character.
`^`	Matches the beginning of a sequence.
`$`	Matches the end of a sequence.
`*`	Matches *zero or more* occurrences of the preceding character or class.
`+`	Matches *at least one* occurrence of the preceding character or class.
`?`	Matches *zero or one* occurrence of the preceding character or class.
`\c`	Matches the character `c` literally, even if it is an operator.

Here are some sample REs:

`[A-Za-z]+`	matches any alphabetic sequence
`[A-Za-z]+[0-9]*`	matches any alphabetic sequence followed by an optional numeric sequence
`[a-z][0-9]?`	matches any lower-case letter followed by an optional digit
`_*`	matches any sequence of any length (even empty ones)
`_+`	matches any sequence of any length (but not empty ones)
`\**`	matches any sequence of asterisks

The following code contains two functions used for regular expression matching: *re-compile* and *re-match*. *Re-compile* compiles a RE to a format that is more suitable for efficient matching. The compiled format is called a CRE (compiled RE):

```
(re-compile RE) => CRE
```

The *re-match* function matches a CRE against a sequence of characters. It returns the subsequence that matches the CRE or **`:f`** if the sequence does not match:

```
(re-match (re-compile '#[a-z]*) '#___abc___) => '#abc
(re-match (re-compile '#[0-9]*) '#___abc___) => :f
```

The matcher uses a first-match and longest-match-first strategy. *First match* means that given multiple potential matches, it returns the first one:

```
(re-match (re-compile '#[a-z]*) '#_abc_def_) => '#abc
```

Longest match first means that each operator matching a subsequence (like `*` and `+`) attempts to match as many characters as possible (this method is also known as *eager* matching):[17]

```
(re-match (re-compile '#x_*x) '#x___x___x) => '#x___x___x
```

Another strategy (which is known as *shortest match first* or *lazy* matching) would attempt to find the shortest possible string matching an RE. Using this approach, the above expression would return `'#x____x`.

As can be seen above, the zenlisp RE matcher uses lists of single-character symbols to represent sequences of characters. Of course, this means that only a limited character set can be used by it, but the underlying principles are the same as, for instance, in the `grep` utility.

Lacking the concept of a "line" of text, the `^` and `$` operators denote the beginning and end of a sequence in this section:

```
(re-match (re-compile '#[a-z]*)     '#12test34) => '#test
(re-match (re-compile '#^12[a-z]*) '#12test34) => '#12test
(re-match (re-compile '#[a-z]*34$) '#12test34) => '#test34
(re-match (re-compile '#^[a-z]*$)  '#12test34) => :f
```

9.3.1 regular expression compilation

The following list defines the character set on which the RE functions will operate. Characters replaced with __ cannot be represented using symbols. The set is basically a subset of ASCII excluding the control character block and the symbols that are reserved for the zenlisp language. Because the interpreter does not distinguish between upper and lower case characters, they are considered to be equal.

```
(define character-set
  '(__ !  "  __ $  %  &  __ __ __ *  +  ,  -  __ /
    0  1  2  3  4  5  6  7  8  9  :  __ <  =  >  ?
    @  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
    p  q  r  s  t  u  v  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  v  w  x  y  z  __ |  __ ~  __))
```

Pair-p is just a short cut.

```
(define (pair-p x) (not (atom x)))
```

The *before-p* predicate checks whether the character *c0* appears before the character *c1* in *character-set*. It will be used to check "–" operators in character classes.

```
(define (before-p c0 c1)
  (letrec
```

17 The longest match first approach is nowadays also called "greedy" matching, but I consider this term to eb unfortunate, because it propagates a destructive mindset that already has caused enough suffering on our planet.

```
    ((lt (lambda (set)
           (cond ((null set) (bottom (list before-b c0 c1)))
                 ((eq c1 (car set)) :f)
                 ((eq c0 (car set)) :t)
                 (t (lt (cdr set)))))))
    (lt character-set)))
```

Make-range adds a new range (from *c0* through *cn*) to the class *cls*, e.g.:

```
(make-range 'a 'f '#9876543210) => '#fedcba9876543210
```

It is used to expand "–" operators in classes.

```
(define (make-range c0 cn cls)
  (letrec
    ((make
       (lambda (c cls)
         (cond ((null c)
                 (bottom 'invalid-symbol-code cn))
               ((eq (car c) cn)
                 (cons (car c) cls))
               (t (make (cdr c)
                        (cons (car c) cls)))))))
    (let ((c (memq c0 character-set)))
      (cond (c (make c cls))
            (t (bottom 'invalid-symbol-code c0))))))
```

The *compile-class* function compiles the character class at the beginning of the argument *in* and conses it to *out*. The *cls* argument holds the operator `[` that will be used to indicate a class in the resulting CRE. This operator will be changed to `]` when compiling complement classes (starting with `[^`). *First* is a flag that is initially set to "true" to indicate that *compile-class* currently processes the first character of the class. It is used to recognize ^ operators. The function returns a list containing the rest of its input as its first member and the compiled class as its second member:

```
(compile-class '#0-9] () '#[ :t) => '(() (#[0123456789))
(compile-class '#^0-9] () '#[ :t) => '(() (#]0123456789))
(compile-class '#0-9]xyz () '#[ :t) => '(#xyz (#[0123456789))
```

When invalid input is passed to *compile-class*, it returns **`:f`**:

```
(compile-class '#0-9 () '#[ :t) => :f  ; missing ]
```

Note that the class operator itself (`[`) has to be consumed by the *caller* of *compile-class*.

```
(define (compile-class in out cls first)
  (cond
    ((null in) :f)
    ((eq '] (car in))
      (list (cdr in) (cons (reverse cls) out)))
    ((and first (eq '^ (car in)))
      (compile-class (cdr in) out '#] :f))
    ((and (not first)
          (not (null (cdr cls)))
          (eq '- (car in))
          (pair-p (cdr in))
          (not (eq '] (cadr in))))
      (let ((c0 (car cls))
            (cn (cadr in)))
        (cond
          ((before-p c0 cn)
            (compile-class (cddr in)
                           out
                           (make-range c0 cn (cdr cls)) :f))
            (t (compile-class (cdr in)
                              out
                              (cons '- cls) :f)))))
    (t (compile-class (cdr in)
                      out
                      (cons (car in) cls) :f))))
```

The *re-compile* function compiles an RE to a compiled RE (CRE). REs map to CREs as follows:

re	**cre**
`[class]`	`'#[class`
`[^class]`	`'#]class`
`pattern*`	`(* pattern)`
`pattern+`	`pattern (* pattern)`
`pattern?`	`(? pattern)`
`^`	`#^`
`$`	`#$`
`\c`	`c`

So, for example:

```
(re-compile '#\*[a-c]+[^d-f]*\*)  =>  '(* #[abc (* #[abc) (* #]def) *)
```

Note that `pattern+` compiles to `pattern (* pattern)`; there is no separate CRE operator implementing `+`. *Re-compile* returns **`:f`** when an invalid RE is passed to it.

```
(define (re-compile re)
  (letrec
     ((compile
       (lambda (in out)
         (cond
           ((not in) :f)
           ((null in) (reverse out))
           (t (cond
                ((eq (car in) '\)
                  (cond ((pair-p (cdr in))
                          (compile (cddr in)
                                   (cons (cadr in) out)))
                        (t :f)))
                ((memq (car in) '#^$_)
                  (compile (cdr in)
                           (cons (list (car in)) out)))
                ((memq (car in) '#*?)
                  (compile (cdr in)
                           (cond ((null out)
                                   (cons (car in) out))
                                 (t (cons (list (car in) (car out))
                                          (cdr out))))))
                ((eq (car in) '+)
                  (compile (cdr in)
                           (cond ((null out)
                                   (cons (car in) out))
                                 (t (cons (list '* (car out)) out)))))
                ((eq (car in) '[)
                  (apply compile
                         (compile-class (cdr in) out '#[ :t)))
                (t (compile (cdr in)
                            (cons (car in) out))))))))
    (compile re ())))
```

9.3.1 regular expression matching

The *match-char* function matches a character (represented by a single-character symbol) against another character or a *class* of characters. For instance:

```
(match-char 'x     'x) => :t
(match-char '#]abc 'x) => :t
(match-char '_     'x) => :t
(match-char '#[123 'x) => :f
```

When the pattern *p* matches the character *c*, it returns truth, else falsity.

```
(define (match-char p c)
  (cond ((eq '_ p)
          :t)
        ((atom p)
          (eq p c))
        ((eq '[ (car p))
          (and (memq c (cdr p)) :t))
        ((eq '] (car p))
          (not (memq c (cdr p))))
        (t :f)))
```

Make-choices generates alternatives for matching subsequences using the `*` operator. For example, when matching the pattern `[a-f]*` against the sequence `abc123`, the following alternatives exist:

```
(make-choices '((* #[abcdef)) '#abc123 ())
=> '((#abc123 ())
     (#bc123 #a)
     (#c123 #ba)
     (#123 #cba))
```

These alternatives are used to *backtrack* when the longest match causes the rest of the RE to mismatch. For instance when matching the RE `a*ab` against the sequence `aaaab`, the `a*` pattern could match `aaaa`, but then the subsequent pattern `ab` would not match `b`, so the match would fail. By backtracking, the matcher would then try to associate `a*` with `aaa`. In this case `ab` is matched against `ab`, so the whole RE matches:

```
(re-match (re-compile '#a*ab) '#aaaab) => '#aaaab
```

Note: The choices created by *make-choices* do include an empty match. The *m* argument of *make-choices* must be `()` initially.

The value returned by *make-choices* contains the rest of the sequence to be matched in its car part and the matched part of the sequence its cadr part. The matched part is returned in reverse order, because it will be passed to *match-cre* later in the process (see below).

```
(define (make-choices p s m)
  (cond
    ((or (null s)
         (not (match-char (cadar p) (car s))))
      (list (list s m)))
    (t (cons (list s m)
             (make-choices p (cdr s) (cons (car s) m))))))
```

The *match-star* function tries the alternatives generated by *make-choices* and finds the longest match that does not make the rest of the RE fail. Note that it returns the sequence matched by the complete remaining part of the CRE passed to it and not just the longest match found for the `*` operator at its beginning:

```
(match-star '((* a) a b) '#aaaab ()) => '#aaaab
```

Match-star reverses the result of *make-choices* because it lists the shortest match first.

```
(define (match-star cre s m)
  (letrec
    ((try-choices
       (lambda (c*)
         (cond ((null c*) :f)
               (t (let ((r (match-cre (cdr cre) (caar c*) (cadar c*))))
                    (cond (r (append (reverse m) r))
                          (t (try-choices (cdr c*))))))))))
    (try-choices (reverse (make-choices cre s ())))))
```

The `match-cre` function matches all characters and operators except for `^`. It matches a compiled RE against the sequence *s*. Matching starts at the beginning of the sequence, so *match-cre* does *not* find occurrences of the RE that occur later in *s*:

```
(match-cre '((* #[ab) c) '#1abc2 ()) => :f
```

An already-matched part may be passed to *matche-cre* using the *m* argument, but it must be in reverse order, because *matche-cre* conses to it and reverses *m* when it is done.

```
(define (match-cre cre s m)
  (cond
    ((null cre)
      (reverse m))
    ((null s)
      (cond ((equal cre '(#$))
              (match-cre () () m))
            ((and (pair-p (car cre))
                  (eq '* (caar cre))
                  (null (cdr cre)))
              ())
            (t :f)))
    ((pair-p (car cre))
      (cond
        ((eq '* (caar cre))
          (match-star cre s m))
        ((eq '? (caar cre))
          (cond ((match-char (cadar cre) (car s))
                  (match-cre (cdr cre) (cdr s) (cons (car s) m)))
                (t (match-cre (cdr cre) s m))))
        ((match-char (car cre) (car s))
          (match-cre (cdr cre) (cdr s) (cons (car s) m)))
        (t :f)))
    ((eq (car cre) (car s))
      (match-cre (cdr cre) (cdr s) (cons (car s) m)))
    (t :f)))
```

Try-matches attempts to match a given RE to each tail of a given sequence, so unlike *match-cre* it also matches occurrences of the RE that begin later in the sequence:

```
(try-matches '((* #[ab) c) '#1abc2) => '#abc
```

Try-matches does not accept empty matches while iterating over the sequence. Only when the sequence has been completely visited without finding a non-empty match, an empty match is tried as a last resort.

```
(define (try-matches cre s)
  (cond ((null s) (match-cre cre s ()))
        (t (let ((r (match-cre cre s ())))
             (cond ((or (not r) (null r))
                     (try-matches cre (cdr s)))
                   (t r))))))
```

The *re-match* function matches the compiled regular expression *cre* against the sequence *s*. When the CRE starts with a ^ operator, *match-cre* is used to match the RE, otherwise *try-matches* is used.

```
(define (re-match cre s)
  (cond ((and (pair-p cre) (equal '#^ (car cre)))
          (match-cre (cdr cre) s ()))
        (t (try-matches cre s))))
```

How are the following REs interpreted? How should they be interpreted? **(Q24)**

```
(re-compile '#[-x])
(re-compile '#[x-])
(re-compile '#[])
(re-compile '#[^])
```

Why are regular expressions compiled and matched separately? Would it not be easier to compile and match them in one step?

The *make-choices* function creates all potential matches for a repetitive pattern, but only one of its results is actually used. Can you modify the code in such a way that it creates the next choice only if the current one does not match?

Can you turn the RE matcher introduced in this section into a shortest-match-first implementation? **(Q25)**

9.4 meta-circular interpretation

A *meta-circular* interpreter for a language **L** is an interpreter that is itself written in **L** and makes use of functions and other facilities provided by the host interpreter instead of re-implementing them. Meta-circular interpreters are capable of interpreting themselves, so whenever a meta-circular interpreter is run, there is an "outer" and and "inner" interpreter, i.e. an interpreter running the (inner) interpreter and an interpreter running a program. This principle is illustrated in figure 7.

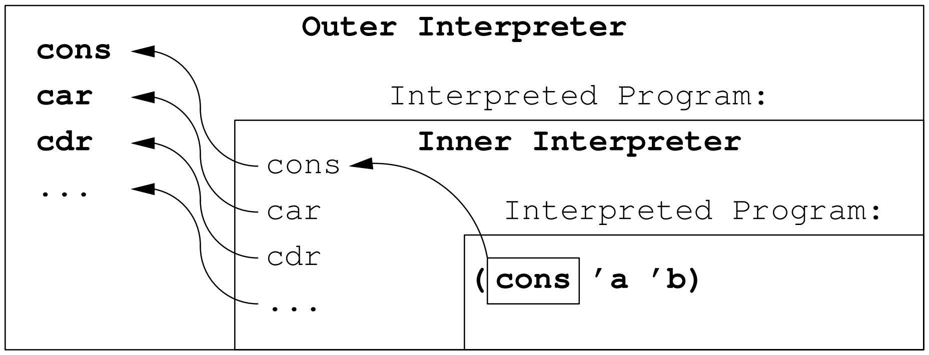

Fig. 7 – meta-circular interpretation

The inner interpreter is a meta-circular interpreter **M** while the outer one may be either another instance of **M** or a "native" interpreter.

This principle illustrates nicely that the question of "interpretation" versus "compilation" is an illusory one: a meta-circular interpreter is interpreted by a native interpreter, which is either interpreted itself or ‘‘compiled’’. When it is compiled, it is in fact interpreted by the CPU. The CPU instructions are interpreted by microcode, microcode is interpreted by transistors. Transistor functions are interpreted by the Laws of Nature. So unless your program is executed directly by the Laws of Nature, it is not really efficient, but probably more comprehensible.

A meta-circular interpreter for zenlisp would not implement functions like **cons** on its own but simply delegate applications of **cons** to the primitive **cons** function of the outer interpreter (see figure 7). Such an interpreter would not employ a parser, either, because zenlisp programs are zenlisp data, so the parser (reader) of the outer interpreter would be used instead.

The *zeval* function discussed in this section implements a meta-circular interpreter for zenlisp (modulo **define** and the meta functions). It accepts a program and an environment as its arguments and returns the normal form of the progam in the given environment:

```
(zeval '(letrec
          ((append
             (lambda (a b)
               (cond ((null a) b)
                     (t (cons (car a)
                          (append (cdr a) b)))))))
          (append '#hello- '#world!))
        (list (cons 'null null)))
=> '#hello-world!
```

Note that the **null** function has to be passed to *zeval* in the environment, because *zeval* implements only the core part of the language. Adding more functions is left as an exercise to the reader.

The code of *zeval* is not as simple as some other meta-circular interpreters that you may find in the lterature, because it interprets tail recursive programs in constant space. Here is the code:

```
(define (zeval x e)
  (letrec
```

The *initial environment* of the interpreter contains the zenlisp keywords, the symbols **:t**, **:f**, and **t**, and a set of functions that cannot be implemented easily without referring to the outer interpreter. The initial environment may be considered a reasonable minimal set of symbols, keywords, and functions that is necessary to build zenlisp.

```
    ((initial-env
       (list (cons 'closure        'closure)
             (cons 't              ':t)
             (cons ':t             ':t)
             (cons ':f             ':f)
             (cons 'and            '(%special . and))
             (cons 'apply          '(%special . apply))
             (cons 'cond           '(%special . cond))
             (cons 'eval           '(%special . eval))
             (cons 'lambda         '(%special . lambda))
             (cons 'let            '(%special . let))
```

```
        (cons 'letrec         '(%special . letrec))
        (cons 'or             '(%special . or))
        (cons 'quote          '(%special . quote))
        (cons 'atom           (cons '%primitive atom))
        (cons 'bottom         (cons '%primitive bottom))
        (cons 'car            (cons '%primitive car))
        (cons 'cdr            (cons '%primitive cdr))
        (cons 'cons           (cons '%primitive cons))
        (cons 'defined        (cons '%primitive defined))
        (cons 'eq             (cons '%primitive eq))
        (cons 'explode        (cons '%primitive explode))
        (cons 'implode        (cons '%primitive implode))
        (cons 'recursive-bind (cons '%primitive recursive-bind))))
```

Value-of finds the value associated with a symbol in an association list and returns it. Unlike **assq**, it yields bottom when the symbol is not contained in the alist or associated with the special value `%void`. This function is used to look up values of variables in environments (which are implemented as alists).

```
(value-of
  (lambda (x e)
    (let ((v (assq x e)))
      (cond ((or (not v) (eq (cdr v) '%void))
              (bottom 'undefined: x))
            (t (cdr v))))))
```

Ev-list evaluates all members of *x* in the environment *e* and returns a list containing their normal forms. It is used to evaluate function arguments.

```
(ev-list
  (lambda (x e)
    (cond ((null x) ())
          ((atom x) (bottom 'improper-list-in-application: x))
          (t (cons (ev (car x) e)
                   (ev-list (cdr x) e))))))
```

Check-args checks whether the argument list *a* has *n* arguments (or >=*n*, if the *more* flag is set). *Wrong-args* reports a wrong argument count. *Args-ok* is a front end to these functions. It is called by primitives to make sure that the correct number of arguments has been supplied.

```
(check-args
  (lambda (a n more)
    (cond ((null n) (or more (null a)))
          ((null a) :f)
          (t (check-args (cdr a)
                         (cdr n)
                         more)))))
(wrong-args
  (lambda (name args)
    (bottom 'wrong-number-of-arguments:
            (cons name args))))
```

```
(args-ok
  (lambda (name a n more)
    (cond ((check-args a n more) :t)
          (t (wrong-args name a)))))
```

The *eval-until* function evaluates the members of the list *a* in the environment *e*. As soon as a member of *a* evaluates to *t/f*, *eval-until* returns **(quote** *t/f***)** immediately. When no member reduces to *t/f*, it returns the last member of *a* (*not* the normal form of that member).

Eval-until is used to implement the **and** and **or** keywords. It returns the last form in unevaluated state because evaluating it in situ would break tail recursion. This will become clear later in the code.

Note that *t/f* must be either **:t** or **:f**. When it is **:t**, it matches *any* “true” value due to the way it is checked in *eval-until*.

```
(eval-until
  (lambda (t/f a e)
    (cond ((null (cdr a)) (car a))
          ((atom a) (bottom 'improper-list-in-and/or: a))
          (t (let ((v (ev (car a) e)))
               (cond ((eq (not v) (not t/f))
                       (list 'quote v))
                     (t (eval-until t/f (cdr a) e))))))))
```

Do-and uses *eval-until* to implement **and**.

```
(do-and
  (lambda (a e)
    (cond ((null a) :t)
          (t (eval-until :f a e)))))
```

The *clause-p* and *do-cond* functions implement **cond**. *Clause-p* checks whether a clause of **cond** is syntactically correct.

Do-cond does a lot of checking first. The semantics of **cond** is implemented in the last clause, which loops through the clauses and returns the expression associated with the first “true” predicate. Like in *do-and* the expression is returned in unevaluated form.

```
(clause-p
  (lambda (x)
    (and (not (atom x))
         (not (atom (cdr x)))
         (null (cddr x)))))
(do-cond
  (lambda (a e)
    (cond ((null a)
            (bottom 'no-default-in-cond))
          ((atom a)
            (bottom 'improper-list-in-cond))
```

```
              ((not (clause-p (car a)))
                (bottom 'bad-clause-in-cond: (car a)))
              (t (let ((v (ev (caar a) e)))
                   (cond (v (cadar a))
                         (t (do-cond (cdr a) e))))))))
```

Do-eval implements **`eval`**. It merely passes its argument to the interpreter.

```
     (do-eval
       (lambda (args e)
         (and (args-ok 'eval args '#i :f)
              (ev (car args) e))))
```

The *lambda-args* function turns an argument list (which may be a list, a dotted list, or even a symbol) into a proper list:

```
(lambda-args '(x y))   => '(x y)
(lambda-args '(x . y)) => '(x y)
(lambda-args 'x)       => '(x)
```

Its code is simple:

```
     (lambda-args
       (lambda (a)
         (cond ((null a) ())
               ((atom a) (list a))
               (t (cons (car a)
                        (lambda-args (cdr a)))))))
```

Add-free-var adds a binding for the free variable *var* to the environment *fenv*. The variable *e* is an environment in which *var* may be bound (it may be unbound as well). Here are some examples:

```
(add-free-var '((a . b)) 'a ()) => '((a . b))
(add-free-var () 'a '((a . x))) => '((a . x))
(add-free-var () 'a ())          => '((a . %void))
```

When the variable already is in *fenv* it is not added again. If it is not in *fenv* but in *e*, the binding of *e* is copied to *fenv*. If *var* is in neither environment, a new binding is created which associates the variable with the special symbol `%void`. This symbol indicates that the variable is not bound to any value.

```
     (add-free-var
       (lambda (fenv var e)
         (cond ((assq var fenv) fenv)
               (t (let ((v (assq var e)))
                    (cond (v (cons v fenv))
                          (t (cons (cons var '%void) fenv))))))))
```

The *capture* function creates a snapshot of all free variables of the current environment. It is used to capture lexical environments when creating closures. *Bound* is a list of bound variables, *x* is the expression whose variables are to be captured, *e* is the currently active environment.

```
    (capture
      (lambda (bound x e)
        (letrec
          ((collect
             (lambda (x free)
               (cond ((null x) free)
                     ((atom x)
                       (cond ((memq x bound) free)
                             (t (add-free-var free x e))))
                     (t (collect (car x)
                                 (collect (cdr x) free)))))))
          (collect x ()))))
```

Do-lambda implements **lambda**. It returns a closure.

```
    (do-lambda
      (lambda (args e)
        (and (args-ok 'lambda args '#ii :f)
             (list 'closure
                   (car args)
                   (cadr args)
                   (capture (lambda-args (car args))
                            (cadr args)
                            e)))))
```

Do-or implements **or**. No surprise here.

```
    (do-or
      (lambda (a e)
        (cond ((null a) :f)
              (t (eval-until :t a e)))))
```

Do-quote implements the **quote** keyword. It simply returns its argument. Note that the interpreter does not explicitly support **'x** in the place of **(quote x)**. This is not necessary, because **'x** is expanded by the reader, so *zeval* never sees the unexpanded form.

```
    (do-quote
      (lambda (args)
        (and (args-ok 'quote args '#i :f)
             (car args))))
```

The *make-env* function creates a new environment from the list of variables (formal arguments) *fa* and the (actual) argument list *aa*. It reduces to bottom when the numbers of arguments do not match. The function handles variadic argument lists correctly:

```
(make-env '(a . b) '(x y z)) => '((a . x) (b . (y z)))
(make-env 'a '(x y z))       => '((a . (x y z)))
```

Make-env is used to bind variables to arguments in function applications.

```
    (make-env
      (lambda (fa aa)
        (cond ((null fa)
                (cond ((null aa) ())
                      (t (bottom 'too-many-arguments))))
              ((atom fa)
                (list (cons fa aa)))
              ((null aa)
                (bottom 'too-few-arguments))
              (t (cons (cons (car fa) (car aa))
                       (make-env (cdr fa) (cdr aa)))))))
```

The *beta* function implements *beta reduction* (see also page 59). It extends the current *outer* environment *e* by the union of the following partial environments:

– the bindings of the variables in *fa* to the arguments in *aa*;
– the lexical environment *lex-env*;
– the current *inner* environment *le* (**l**ocal **e**nvironment).

It also applies the *fix* function to the new local bindings formed by *fa* and *aa*. When *fix*=**`id`**, then *beta* implements ordinary beta reduction as in **`lambda`** and **`let`**. When *fix*=**`recursive-bind`**, it implements **`letrec`**.

Finally *beta* evaluates the expression *expr*. It passes both the inner (*e*) and outer environment (*le*) to *ev2*. This is done in order to implement tail recursion.

```
    (beta
      (lambda (expr fa aa lex-env e le fix)
        (ev2 expr e (append (fix (make-env fa aa)) lex-env le))))
```

Do-let/rec implements **`let`** and **`letrec`**. It merely extracts the variables and arguments from the constructs and passes them to *beta*. *Le* and *e* are the current inner and outer environments. The *fix* parameter is explained above.

The *binding-p* helper makes use of the fact that clauses have the same syntax as bindings.

```
    (binding-p
      (lambda (x)
        (clause-p x)))
    (do-let/rec
      (lambda (args e le fix)
        (cond ((not (args-ok 'let/letrec args '#ii :f)) :f)
              ((not (apply and (map binding-p (car args))))
                (bottom 'bad-let/letrec-syntax: (car args)))
              (t (let ((formals (map car (car args)))
                       (actuals (map cadr (car args))))
                   (beta (cadr args)
                         formals
                         (ev-list actuals le)
                         ()
                         e le fix))))))
```

Apply-fn applies the function *fn* to the formal arguments *args*. The arguments already are in their normal forms at this point. The environments *e* and *le* are merely passed through.

```
     (apply-fn
       (lambda (fn args e le)
         (cond ((eq (car fn) '%primitive)
                 (apply (cdr fn) args))
               ((eq (car fn) '%special)
                 (apply-special (cdr fn) args e le))
               ((eq (car fn) 'closure)
                 (beta (caddr fn)
                       (cadr fn)
                       args
                       (cadddr fn)
                       e le id))
               (t (bottom 'application-of-non-function: fn)))))
```

The *make-args* function creates an argument list for **apply**. Because **apply** itself is variadic, it collects the optional arguments and makes sure that the last one is a list:

```
(make-args '(a b c '(d e))) => '#abcde
(make-args '(a b c d))      => bottom
```

Here comes the code:

```
     (make-args
       (lambda (a)
         (cond ((null (cdr a))
                 (cond ((atom (car a))
                         (bottom 'improper-argument-list:
                                 (car a)))
                       (t (car a))))
               (t (cons (car a) (make-args (cdr a)))))))
```

Apply-special interprets the keywords of zenlisp (except for **define**). It applies the pseudo function *fn* to the arguments *args*. Because *fn* is a pseudo function, the arguments are unevaluated and *not* in their normal forms.

Note that some of the cases of *apply-special* pass the value returned by a special form handler to *ev2* and some simply return it. This is because some special form handlers (for **and**, **cond**, etc) return expressions that still have to be reduced to their normal forms while others return normal forms immediately (like the handlers for **lambda**, **quote**, etc).

Still others (like the handlers for **let**, **letrec** and **apply**) call *ev2* themselves, so they recurse indirectly. Because handlers like *do-let/rec* are called in tail positions, recursion still happens in constant space.

```
     (apply-special
       (lambda (fn args e le)
         (cond ((eq fn 'and)
                 (ev2 (do-and args le) e le))
               ((eq fn 'apply)
                 (let ((args (ev-list args le)))
                   (and (args-ok 'apply args '#ii :t)
                        (apply-fn (car args)
                                  (make-args (cdr args))
                                  e
                                  e))))
               ((eq fn 'cond)
                 (ev2 (do-cond args le) e le))
               ((eq fn 'eval)
                 (ev2 (do-eval args le) e le))
               ((eq fn 'lambda)
                 (do-lambda args le))
               ((eq fn 'let)
                 (do-let/rec args e le id))
               ((eq fn 'letrec)
                 (do-let/rec args e le recursive-bind))
               ((eq fn 'or)
                 (ev2 (do-or args le) e le))
               ((eq fn 'quote)
                 (do-quote args))
               (t (bottom 'internal:bad-special-operator: fn)))))
```

These predicates check whether an object is a function or a special form handler.

```
(function-p
  (lambda (x)
    (or (eq (car x) '%primitive)
        (eq (car x) 'closure))))
(special-p
  (lambda (x)
    (eq (car x) '%special)))
```

Ev2 reduces the expression *x* to its normal form in the environment *le*. The argument *e* holds the current *outer* environment. Initially, *e* equals *le*.

In order to find out what to do with a list, *ev2* reduces the car part of the list in *le*. If the resulting normal form *f* is a function, it evaluates the function arguments (the cdr part of the list) in *le* and then discards the inner environment by setting *new-e*=*e*. When *f* is a special form handler, function arguments are not evaluated and the inner environment is kept.

Abandoning the inner environment after evaluating function arguments implements tail recursion.

```
(ev2
  (lambda (x e le)
    (cond
      ((null x) ())
      ((atom x) (value-of x le))
```

```
          (t (let ((f (ev (car x) le)))
               (cond ((eq f 'closure) x)
                     ((atom f)
                       (bottom 'application-of-non-function: f))
                     (t (let ((args (cond ((function-p f)
                                           (ev-list (cdr x) le))
                                          (t (cdr x))))
                              (new-e (cond ((special-p f) le)
                                           (t e))))
                          (apply-fn f args e new-e)))))))))
```

This is just an abbreviation for the case *e=le*.

```
    (ev (lambda (x e)
          (ev2 x e e))))
```

The environment passed to *zeval* is attached to the initial environment before starting the interpreter:

```
   (ev x (append e initial-env))))
```

Can you rewrite *zeval* in such a way that it actually can interpret itself?

Primitives are just passed through to the host interpreter, e.g. **`cons`** is interpreted by the code implementing **`cons`** in the outer interpreter. Why is it not possible to delegate evaluation of special forms to the special form handlers of the outer interpreter? **(Q26)**

Zeval uses symbols like `%special` and `%void` to tag special values. Could this detail cause any trouble when programs to be interpreted by *zeval* contained such symbols? What can you do about it? **(Q27)**

10. mexprc – an m-expression compiler

This chapter will use some of the techniques introduced in the previous chapter to implement a compiler for a full programming language.

M-expressions (*meta expressions*) form the syntax that was originally intended for LISP. In the original design, *S-expressions* were only used for the representation of data, while M-expressions were used to write programs. An *S-expression* is basically equal to a zenlisp form.

Here is a quote from the ACM paper "History of LISP" [18] by John McCarthy that explaines why S-expressions were eventually used in the place of M-expressions:

> S.R. Russel noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter. The unexpected appearance of an interpreter tended to freeze the form of the language [...]
> The project of defining M-expressions precisely and compiling them or at least translating them to S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised.

So the syntax of M-expressions was never specified precisely. However, many LISP texts make use of M-expressions, so enough information can be gathered from them to construct a language that probably comes close to what M-expressions would have been. This chapter defines a BNF grammar for M-expressions and implements a compiler that translates M-expressions to S-expressions (zenlisp programs).

10.1 specification

M-expressions are similar to the infix notation that is employed by FORTRAN, C, Java, and other programming languages. Numbers represent themselves and the usual operators are used to express math operations such as subtraction, multiplication, etc. Due to the lack of suitable characters in the ASCII character set, logical AND and OR is represented by the sequences `/\` and `\/` respectively and the right arrow is written as `->`.

"Real" M-expressions would make use of the characters `(`, `)`, and `;`, but the MEXPRC compiler cannot implement them in this way, because these characters cannot be contained as data in zenlisp programs. Hence the specification has to be adjusted a bit:

18 The full paper can be found at McCarthy's homepage:
`http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`

– expression grouping is done by `[` and `]` rather than `(` and `)`;
– literal lists are delimited by << and >> rather than `(` and `)`;
– function arguments are separated using `,` instead of `;`;
– the conditional operator is written `[a->b:c]` rather than `[a->b;c]`;
– constants are prefixed with `%` instead of using upper case.

Here are some sample M-expressions and their corresponding S-expressions:

```
cons[a,b]                 (cons a b)

a::b                      (cons a b)

%a::%b                    (cons 'a 'b)

%a::<<b,c,d>>             (cons 'a '(b c d))

append[a,b]               (append a b)

a++b                      (append a b)

a*b-c/d                   (- (* a b) (/ c d))

[a+b]*c                   (* (+ a b) c)

[a/\b-> c: d]             (cond ((and a b) c) (t d))

f[x] := x^2               (define (f x) (expt x 2))

lambda[[x] cons[x,x]]     (lambda (x) (cons x x))

lambda[[x] x][%a]         ((lambda (x) x) 'a)

not[x] :=                 (define (not x)
  [x-> false: true]         (cond (x :f) (t :t)))

fact[x] :=                (define (fact x)
  [x=0                      (cond ((= x 0)
     -> 1:                          1)
   fact[x-1]*x]                   (t (* (fact (- x 1)) x))))

f[x] := g[x]              (define (f x)
  where g[x] := h[x]        (letrec ((g (lambda (x) (h x)))
  and h[x] := x                      (h (lambda (x) x)))
                              (g x)))
```

The complete syntax of the source language accepted by MEXPRC is described in the BNF grammar following below. The grammar uses the *concatenation operator* `&` which is not typically found in BNF grammars. Rules of the form

```
a* := a | a & a*
```

match only sequences of `a`s that do not contain any blank characters in between, e.g. the above rule matches

```
aaaaaa
```

but not

```
a a a a a a
```

Concatenation operators (`&`) bind stronger than OR operators (`|`). They are used to introduce token classes like sequences of digits representing numbers or sequences of characters representing symbols.

Some rules of the grammar are annotated with S-expressions that indicate what the sentence matched by the rule is to be translated to. Such annotations are common for specifying the semantics of a language semi-formally. For instance the rule

```
concatenation :=
  factor '::' concatenation   ; (cons factor concatenation)
```

states that each sentence of the above form denotes an application of **`cons`**. Like zenlisp comments, annotations are introduced by a semicolon and extend up to the end of the current line.

10.1.1 annotated grammar

Note: non-terminals have no angle brackets in this grammar. All terminals are enclosed in apostrophes.

```
mexpr := definition
       | expression

numeric-char := '0' | ...|  '9'

symbolic-char := 'a' | ...|  'z' | '_'

number := numeric-char
        | number & numeric-char

symbol := symbolic-char
        | symbol & symbolic-char

list-member := symbol
             | list

list-members := list-member
              | list-member list-members

list := '<<' list-members '>>'   ; (quote (list-members))
      | '<<' '>>'                ; ()

list-of-expressions := expression
                     | expression ',' list-of-expressions
```

```
list-of-symbols := symbol
                 | symbol ',' list-of-symbols

cases := case
       | case ':' cases

case := expression '->' expression

factor :=
    number                             ; number
  | symbol                             ; variable
  | '%' symbol                         ; (quote symbol)
  | 'true'                             ; :t
  | 'false'                            ; :f
  | '-' factor                         ; (- factor)
  | symbol '[' list-of-expressions ']' ; (symbol list-of-expressions)
  | symbol '[' ']'                     ; (symbol)
  | '[' expression ']'                 ; expression
  | '[' cases ':' expression ']'       ; (cond cases (t expression))
                                       ; where cases
                                       ; = ((expression expression) ...)
                                       ; as in "case"
  | lambda                             ; (lambda ...)
  | lambda '[' list-of-expressions ']' ; ((lambda ...) list-of-expressions)
  | lambda '[' ']'                     ; ((lambda ...))

lambda :=
    'lambda' '[' '[' list-of-symbols ']' expression ']'
      ; (lambda (list-of-symbols) expression)
  | 'lambda' '[' '[' ']' expression ']'
      ; (lambda () expression)

concatenation :=
    factor
  | factor '::' concatenation   ; (cons factor concatenation)
  | factor '++' concatenation   ; (append factor concatenation)

power := concatenation
       | concatenation '^' power   ; (expt concatenation power)

term := power
      | term '*' power      ; (* term power)
      | term '/' power      ; (/ term power)
      | term '//' power     ; (quotient term power)
      | term '\\' power     ; (remainder term power)

sum := term
     | sum '+' term   ; (+ sum term)
     | sum '-' term   ; (- sum term)
```

```
predicate := sum '=' sum    ; (= sum sum)
           | sum '<>' sum   ; (not (= sum sum))
           | sum '<' sum    ; (< sum sum)
           | sum '>' sum    ; (> sum sum)
           | sum '<=' sum   ; (<= sum sum)
           | sum '>=' sum   ; (>= sum sum)
           | sum

conjunction :=
    predicate
  | conjunction '/\' predicate   ; (and conjunction predicate)

disjunction :=
    conjunction
  | disjunction '\/' conjunction   ; (or disjunction conjunction)

expression := disjunction

definition :=
    simple-definition
  | simple-definition 'where' definition-list
      ; (define ... (letrec definition-list ...))
      ; where definition-list
      ;         = ((symbol (lambda (list-of-symbols) expression)) ...)
      ;         | ((symbol (lambda () expression)) ...)
      ; as in "simple-definition"

simple-definition :=
    symbol '[' list-of-symbols ']' := expression
      ; (define (symbol list-of-symbols) expression)
  | symbol '[' ']' := expression
      ; (define (symbol) expression)

definition-list := simple-definition
                 | simple-definition 'and' definition-list
```

10.2 implementation

We will dive right into the code.

The rational math package is required because of the / operator.

```
(require '~rmath)
```

These classes contain the characters used to compose symbols and numbers:

```
(define symbol-class '#abcdefghijklmnopqrstuvwxyz_)

(define number-class '#0123456789)
```

Symbol-p and *number-p* check whether a character (represented by a single-character symbol) belongs to a specific class:

```
(define (symbol-p x) (and (memq x symbol-class) :t))

(define (number-p x) (and (memq x number-class) :t))
```

10.2.1 lexical analysis

The *prefix->infix* parser [page 117] used lists of symbols to represent input programs. Because M-expressions may be longer than a few characters and may span multiple lines of input, this representation is not suitable for M-expressions, though.

Instead of lists of single-character symbols, MEXPRC will use lists of symbols of variable length. Each symbol may contain one or multiple tokens. Here is a sample M-expression in MEXPR notation:

```
'(  fact[x] :=
      [x=0-> 1:
             fact[x-1]*x]  )
```

This list contains the following symbols

```
fact[x]
:=
[x=0->
1:
fact[x-1]*x]
```

Each of the symbols contains at least one *token*. *Token* is just another word for *lexeme*. It is quite common in compiler texts. These are the individual tokens contained in the above list:

```
fact [ x ]
:=
[ x = 0 ->
1 :
fact [ x - 1 ] * x ]
```

A symbol containing some tokens is called a *fragment*, because it holds a fragment of an input program. Each M-expression program is represented by a sequence of fragments.

The first stage of the MEXPRC compiler scans such a sequence of fragments and decomposes them into individual tokens. This process is called *lexical analysis*. The output of this stage is a sequence of tokens:

```
(tokenize '(fact[x] := [x=0-> 1: fact[x-1]*x]))
=> '(fact [ x ] := [ x = 0 -> 1 : fact [ x - 1 ] * x ])
```

Fragments are exploded when they are are encountered in the input stream for the first time:

```
(define (explode-on-demand fragment)
  (cond ((atom fragment) (explode fragment))
        (t fragment)))
```

Extract-class extracts a sequence matching a character class from the front of a fragment.

```
(define (extract-class fragment class-p)
  (letrec
    ((input
       (explode-on-demand fragment))
     (x-class
       (lambda (input sym)
         (cond ((null input)
                 (list (reverse sym) input))
               ((class-p (car input))
                 (x-class (cdr input)
                          (cons (car input) sym)))
               (t (list (reverse sym) input))))))
    (x-class input ())))
```

The following functions extract symbols and numbers from the current fragment. Like most lexical analysis functions, they return a list containing the extracted token and the remaining input, both in exploded form:

```
(extract-symbol 'abc+def) => '(#abc #+def)
```

The functions are implemented on top of *extract-class*:

```
(define (extract-symbol fragment)
  (extract-class fragment symbol-p))

(define (extract-number fragment)
  (extract-class fragment number-p))
```

Extract-char simply extracts the leading character from the input:

```
(define (extract-char fragment)
  (let ((input (explode-on-demand fragment)))
    (list (list (car input)) (cdr input))))
```

The *extract-alternative* function extracts a single- or double-character token from the head of a fragment. If the second character of the fragment is contained in the *alt-tails* argument, a two-character token is extracted and else a single character is extracted. For instance:

```
(extract-alternative '>=xyz '#>=) => '(#>= #xyz)
(extract-alternative '>-xyz '#>=) => '(#> #-xyz)
```

The function is used to extract operator symbols.

```
(define (extract-alternative fragment alt-tails)
  (let ((input (explode-on-demand fragment)))
    (cond ((null (cdr input))
            (extract-char input))
          ((memq (cadr input) alt-tails)
            (list (list (car input) (cadr input))
                  (cddr input)))
          (t (extract-char input)))))
```

Extract-token extracts any kind of token by dispatching its input to one of the above functions. When the first character of the current fragment cannot be identified, the function signals a syntax error.

```
(define (extract-token fragment)
  (let ((input (explode-on-demand fragment)))
    (let ((first (car input)))
      (cond ((eq first '[)
              (extract-char input))
            ((eq first '])
              (extract-char input))
            ((eq first ',)
              (extract-char input))
            ((eq first '%)
              (extract-char input))
            ((eq first ':)
              (extract-alternative input '#:=))
            ((eq first '+)
              (extract-alternative input '#+))
            ((eq first '-)
              (extract-alternative input '#>))
            ((eq first '*)
              (extract-char input))
            ((eq first '=)
              (extract-char input))
            ((eq first '<)
              (extract-alternative input '#<>=))
            ((eq first '>)
              (extract-alternative input '#>=))
            ((eq first '/)
              (extract-alternative input '#/\))
            ((eq first '\)
              (extract-alternative input '#/\))
            ((eq first '^)
              (extract-char input))
            ((symbol-p first)
              (extract-symbol input))
            ((number-p first)
              (extract-number input))
            (t (bottom 'syntax 'error 'at input))))))
```

These are just some more comprehensible names for members of the intermediate format used during lexical analysis:

```
(define frag car)       ; fragment of input
(define rest cdr)       ; rest of input
(define restfrag cadr)  ; fragment of rest of input
(define restrest cddr)  ; rest of rest of input
```

The *next-token* function extracts the first token of the first fragment of a fragment list. If the first fragment is empty, it removes it and advances to the next fragment.

```
(define (next-token source)
  (cond ((null (frag source))
          (cond ((null (rest source)) ())
                (t (let ((head (extract-token (restfrag source))))
                     (cons (implode (frag head))
                           (cons (restfrag head)
                                 (restrest source))))))))
        (t (let ((head (extract-token (frag source))))
             (cons (implode (frag head))
                   (cons (restfrag head)
                         (rest source)))))))
```

The *tokenize* function forms the front end of the *scanner* of the MEXPR compiler. The *scanner* is the part of a compiler that performs lexical analysis. *Tokenize* accepts a source program and emits a list of separate tokens.

```
(define (tokenize source)
  (letrec
    ((tok (lambda (src tlist)
            (let ((new-state (next-token src)))
              (cond ((null new-state) (reverse tlist))
                    (t (tok (cdr new-state)
                            (cons (car new-state)
                                  tlist))))))))
    (tok source ())))
```

10.2.2 syntax analysis and code synthesis

Syntax analysis and *code synthesis* are interleaved in the following code. Zenlisp expressions are synthesized *while* performing syntax analysis.

The parser is the controlling instance of the MEXPRC compiler: it reads input through the scanner and emits code through the synthesizer (which in this case is not even a separate stage). This is why this approach is called *syntax-directed compilation.*

The parsing stage of the compiler accepts input in the form of a token list (as generated by the scanner) and emits a zenlisp expression:

```
(mexpr-compile (f [ x ] := 2 ^ x) => (define (f x) (expt '#2 x))
```

Most functions of the syntax analysis stage of the compiler process partially translated programs of the form

(*S-expression token-list*)

where S-expression is a zenlisp program under construction and token-list is the not yet translated rest of the input program. Initially the S-expression part is empty and the token-list part contains the full tokenized input program. During compilation, data is moved from the token-list to the S-expression part and when compilation finishes successfully, the token-list part is empty and the S-expression part contains a complete zenlisp expression.

The *parse-term* function, for instance, parses and synthesizes a term:

```
(parse-term '(() #a*b+c))  =>  '((* a b) '#+c)
```

Initially, the synthesized expression is () and the input program is '#a*b+c. *Parse-term* removes the term '#a*b and generates a program structure containing the synthesized expression '(* a b) and the "rest" '#+c.

Make-prog composes a program structure:

```
(define (make-prog sexpr tlist) (list sexpr tlist))
```

These functions are used to access the individual parts of a program structure:

```
(define s-expr-of car)    ; S-expression built so far

(define rest-of cadr)     ; Not yet translated rest of program
```

Has the end of the input program been reached?

```
(define (end-of p) (null (rest-of p)))
```

This function delivers the current input token or (), if input is exhausted.

```
(define (first-of-rest p)
  (cond ((end-of p) ())
        (t (caadr p))))
```

Rest-of-rest delivers the input program with its first token removed. It is used to advance to the next input token.

```
(define (rest-of-rest p)
  (cond ((end-of p) ())
        (t (cdadr p))))
```

Look-ahead and *rest-of-look-ahead* are similar to *first-of-rest* and *rest-of-rest*, but produce the second token in the input and the rest following the second token respectively:

```
(first-of-rest      '(() #a+b*c)) => 'a
(rest-of-rest       '(() #a+b*c)) => '#+b*c
(look-ahead         '(() #a+b*c)) => '+
(rest-of-look-ahead '(() #a+b*c)) => '#b*c
```

The second character in the input is called the *look-ahead* token. It is used to figure out what kind of input is following when the first token is ambiguous. For instance, a symbol name may or may not be followed by an opening parenthesis. When there is an opening parentheses, the symbol is part of a function application (or definition) and otherwise it is a reference to a variable:

```
'(sym + sym)
'(sym [ arg ])
```

In this case, the look-ahead token can be used to delegate further analysis to the appropriate procedure.

```
(define (look-ahead p)
  (cond ((end-of p) ())
        ((null (rest-of-rest p)) ())
        (t (car (rest-of-rest p)))))

(define (rest-of-look-ahead p)
  (cond ((end-of p) ())
        ((null (rest-of-rest p)) ())
        (t (cdr (rest-of-rest p)))))
```

Extract first char of a token

```
(define (first-char x) (car (explode x)))
```

Quoted quotes a form.

```
(define (quoted x) (list 'quote x))
```

Parse-list is the first function that actually performs some syntax analysis. It accepts an M-expression representing a literal list and returns a quoted list in S-expression form, for example:

```
<< a, b, <<c>>, d>>  —>  (quote (a b (c) d))
```

(The arrow denotes the transformation of an M-expression to an S-expression in this section.)

The embedded *plist* function, which performs the actual work, has four parameters with the following meanings: *tls* is the input token list, *lst* is the output list constructed so far, *skip* is used to skip over commas, and *top* is a flag indicating whether we are currently parsing a top level list (as opposed to a nested list).

The function recurses to parse embedded lists.

```
(define (parse-list tlist)
  (letrec
    ((plist
       (lambda (tls skip lst top)
         ; tls  = input
         ; skip = skip next token (commas)
         ; lst  = output
         ; top  = processing top level list
         (cond ((eq (car tls) '>>)
                 (cond (top (make-prog (quoted (reverse lst))
                                        (cdr tls)))
                       (t (make-prog (reverse lst)
                                      (cdr tls)))))
               ((eq (car tls) '<<)
                 (let ((sublist (plist (cdr tls) :f () :f)))
                   (plist (rest-of sublist)
                          :t
                          (cons (car sublist) lst)
                          top)))
               (skip
                 (cond ((eq (car tls) ',)
                     (plist (cdr tls) :f lst top))
                   (t (bottom ', 'expected 'at tls))))
               (t (plist (cdr tls)
                         :t
                         (cons (car tls) lst)
                         top))))))
    (plist tlist :f () :t)))
```

The following function reports an unexpected end of input. This is just an abbreviation that is being introduced because an unexpected EOT (end of text) is a common condition.

```
(define (unexpected-eot)
  (bottom 'unexpected-end-of-input))
```

The *parse-actual-args* function parses a list of function arguments (a list of expressions) and returns an equivalent list of S-expressions:

```
[a+b, c*d]  —>  ((+ a b) (* c d))
```

It is used in applications of named functions and lambda functions.

```
(define (parse-actual-args  tlist)
  (letrec
    ((pargs
       (lambda (tls skip lst)
         (cond ((null tls) (unexpected-eot))
               ((eq (car tls) '])
                 (make-prog (reverse lst) (cdr tls)))
               (skip
                 (cond ((eq (car tls) ',)
                         (pargs (cdr tls) :f lst))
                       (t (bottom ', 'expected 'at tls))))
               (t (let ((expr (parse-expr tls)))
                    (pargs (rest-of expr)
                           :t
                           (cons (car expr) lst))))))))
    (pargs tlist :f ())))
```

Parse-formal-args parses the formal argument list of a function (the variables of a function):

```
[a,b,c]  -->  (a b c)
```

It is similar to *parse-actual-args* but accepts a list of symbols rather than a list of expressions.

```
(define (parse-formal-args  tlist)
  (letrec
    ((pargs
       (lambda (tls skip lst)
         (cond ((null tls) (unexpected-eot))
               ((eq (car tls) '])
                 (make-prog (reverse lst) (cdr tls)))
               (skip
                 (cond ((eq (car tls) ',)
                         (pargs (cdr tls) :f lst))
                       (t (bottom ', 'expected 'at tls))))
               ((symbol-p (first-char (car tls)))
                 (pargs (cdr tls) :t (cons (car tls) lst)))
               (t (bottom 'symbol 'expected 'at tls))))))
    (pargs tlist :f ())))
```

Parse-fun-call parses a function application:

```
f[a,b,c]  -->  (f a b c)
```

It is used to parse applications of named functions exclusively.

```
(define (parse-fun-call program)
  (let ((function (first-of-rest program))
        (args (parse-actual-args (rest-of-look-ahead program))))
    (make-prog (append (list function)
                       (s-expr-of args))
               (rest-of args))))
```

Parse-lambda-args parses the formal argument list of a lambda function, if it follows in the input stream. If no argument list follows, report an error.

```
(define (parse-lambda-args program)
  (cond ((eq (first-of-rest program) '[)
          (parse-formal-args (rest-of-rest program)))
        (t (bottom 'argument 'list 'expected 'in 'lambda[]))))
```

Create a lambda function.

```
(define (make-lambda args term)
  (list 'lambda args term))
```

The *parse-lambda-app* function parses the application of a lambda function:

[a_1 ... a_n] --> ((lambda ...) a_1 ... a_n)

When this function is entered, a leading lambda function has just been parsed and its code already is in the S-expression part of the program structure passed to it:

```
(parse-lambda-app '((lambda (x) x) #[y]+z))  =>  '(((lambda (x) x) y) #+z)
```

Parse-lambda-app is invoked when a lambda function in the source program is followed by an opening parenthesis.

```
(define (parse-lambda-app program)
  (let ((args (parse-actual-args (rest-of-rest program))))
    (make-prog (append (list (s-expr-of program))
                       (s-expr-of args))
               (rest-of args))))
```

Parse-lambda parses a lambda function:

lambda[[v_1 ... v_n] term] —> (lambda (v_1 ... v_n) term)

It uses the look-ahead token for convenience. No actual look ahead is necessary at this point.

```
(define (parse-lambda program)
  (cond ((neq (look-ahead program) '[)
          (bottom '[ 'expected 'after 'lambda))
        (t (let ((args (parse-lambda-args
                         (make-prog
                           ()
                           (rest-of-look-ahead program)))))
          (let ((term (parse-expr (rest-of args))))
            (cond ((neq (first-of-rest term) '])
                    (bottom 'missing 'closing '] 'in 'lambda[]))
                  (t (make-prog
                       (make-lambda (s-expr-of args)
                                    (s-expr-of term))
                       (rest-of-rest term))))))))))
```

Create a clause for **cond**:

```
(define (make-case pred expr) (list pred expr))
```

Parse-cases parses the cases of a conditional expression and generates a set of clauses for **cond**:

```
a->b: c->d: e  —>  ((a b) (c d) (t e))
```

Note that the function parses just the cases without the brackets that enclose a conditional M-expression.

```
(define (parse-cases program)
  (letrec
    ((pcases
       (lambda (prog cases)
         (let ((pred (parse-disj (make-prog () prog))))
           (cond
             ((neq (first-of-rest pred) '->)
               (make-prog (cons (make-case 't (s-expr-of pred))
                                cases)
                          (rest-of pred)))
             (t (let ((expr (parse-expr (rest-of-rest pred))))
                  (cond
                    ((eq (first-of-rest expr) ':)
                      (pcases (rest-of-rest expr)
                              (cons (make-case (s-expr-of pred)
                                               (s-expr-of expr))
                                    cases)))
                    (t (bottom ': 'expected 'in 'conditional 'before
                               (rest-of expr))))))))))
    (let ((case-list (pcases (rest-of program) ())))
      (make-prog (reverse (s-expr-of case-list))
                 (rest-of case-list)))))
```

Make-cond-expr creates a **cond** expression from a list of cases. When there is only one case (the default), skip **cond** and just synthesize the expression constituting that case.

This is a necessary extension and not just a performance hack, because *parse-cond-expr* (below) exploits the fact that a conditional expression with just a default case is syntactically equal to grouping: See *parse-cond-expr* for details.

```
(define (make-cond-expr cases)
  (cond ((null (cdr cases))
          (cadar cases))
        (t (cons 'cond cases))))
```

Parse-cond-expr parses a conditional expression and generates an equivalent **cond** expression:

$[p_1\text{-> } x_1\text{: } p_2\text{-> } x_2\text{: ... } x_n] \longrightarrow (\text{cond } (p_1\ x_1)\ (p_2\ x_2)\ \ldots\ (\text{t } x_n))$

In fact, it merely skips over the opening square bracket, delegates analysis of the cases to *parse-cases* (above) and then makes sure that there is a delimiting closing bracket.

Parse-cond-expr handles both conditional expressions and grouping (because they are syntactically equal):

```
[pred->expr: expr]  —>  (cond (pred expr) (t expr))
[expr]              —>  (cond (t expr))             —>  expr
```

The redundant **cond** is removed by *make-cond-expr* (above).

```
(define (parse-cond-expr program)
  (let ((cond-expr
          (parse-cases
            (make-prog () (rest-of-rest program)))))
    (cond ((neq (first-of-rest cond-expr) '])
            (bottom '] 'expected 'at 'end 'of
                     'conditional 'expression))
          (t (make-prog
               (make-cond-expr (s-expr-of cond-expr))
               (rest-of-rest cond-expr))))))
```

Because *parse-cond-expr* has two functions, it should have two names that reflect these functions:

```
(define parse-grouped-expr parse-cond-expr)
```

The *parse-factor* function accepts a single factor of an M-expression and generates an equivalent S-expression. When the token at the beginning of the source part of the program structure *program* is not a valid factor, a syntax error is reported.

Transformations are inlined in the below code.

```
(define (parse-factor program)
  (let ((first (first-char (first-of-rest program))))
    (cond ((null first)
            (unexpected-eot))

; nil  —>  ()

          ((eq (first-of-rest program) 'nil)
            (make-prog () (rest-of-rest program)))

; true  —>  :t

          ((eq (first-of-rest program) 'true)
            (make-prog :t (rest-of-rest program)))

; false —>  :f

          ((eq (first-of-rest program) 'false)
            (make-prog :f (rest-of-rest program)))

; lambda[[v ...] x]      —>  (lambda (v ...) x)
; lambda[[v ...] x][a]   —>  ((lambda (v ...) x) a)

          ((eq (first-of-rest program) 'lambda)
            (let ((lambda-term (parse-lambda program)))
              (cond ((eq (first-of-rest lambda-term) '[)
                      (parse-lambda-app lambda-term))
                    (t lambda-term))))

; symbol          —>  symbol
; symbol[x ...]   —>  (symbol x ...)

          ((symbol-p first)
            (cond ((eq (look-ahead program) '[)
                    (parse-fun-call program))
                  (t (make-prog (first-of-rest program)
                                (rest-of-rest program)))))
; number  -->  (quote #number)

          ((number-p first)
            (make-prog (quoted
                         (explode
                           (first-of-rest program)))
                       (rest-of-rest program)))

; << element, ... >>  -->  (quote (element ...))

          ((eq (first-of-rest program) '<<)
            (parse-list (rest-of-rest program)))

; %symbol  -->  (quote symbol)

          ((eq first '%)
            (cond ((symbol-p (first-char (look-ahead program)))
                    (let ((rhs (parse-factor
                                 (make-prog
                                   ()
                                   (rest-of-rest program)))))
                      (make-prog (quoted (s-expr-of rhs))
                                 (rest-of rhs))))
                  (t (bottom 'symbol 'expected 'after '%: program))))

; [expression]  -->  expression

          ((eq first '[)
            (parse-grouped-expr program))

; -factor  -->  (- factor)

          ((eq first '-)
            (let ((rhs (parse-factor
                         (make-prog
                           ()
                           (rest-of-rest program)))))
              (make-prog (list '- (s-expr-of rhs))
                         (rest-of rhs))))

          (t (bottom 'syntax 'error 'at (rest-of program))))))
```

The *parse-binary* function parses all kinds of left-associative binary operators:

x op y op z --> (fun_{op} (fun_{op} x y) z)

The function expects an association list of operators and functions in the *ops* argument, where each key is an operator symbol and each value is the name of a function implementing the operator. Examples will be given below.

The *parent-parser* argument will be bound to a function parsing the factors of the operators. Each factor of a chain of operations may contain operations of a higher precedence. These higher-precedence operations are handled by *parent-parser*.

This function is a generalization of the *sum* and *term* functions of the *infix->prefix* parser introduced on page 113.

```
(define (parse-binary program ops parent-parser)
  (letrec
    ((lhs (parent-parser program))
     (collect
       (lambda (expr tlist)
         (let ((op (cond ((null tlist) :f)
                         (t (assq (car tlist) ops)))))
           (cond ((null tlist)
                   (make-prog expr ()))
                 (op (let ((next (parent-parser
                                   (make-prog () (cdr tlist)))))
                       (collect (list (cdr op) expr (s-expr-of next))
                                (rest-of next))))
                 (t (make-prog expr tlist)))))))
    (collect (car lhs) (rest-of lhs))))
```

Parse-binary-r is like *parse-binary*, but parses and synthesizes right-asscociative operators:

$x\ op\ y\ op\ z \quad \longrightarrow \quad (\mathrm{fun}_{op}\ x\ (\mathrm{fun}_{op}\ y\ z))$

It is a generalization of the *power* function of the *infix->prefix* parser [page 113].

```
(define (parse-binary-r program ops parent-parser)
  (let ((lhs (parent-parser program)))
    (let ((op (cond ((null (rest-of lhs)) :f)
                    (t (assq (first-of-rest lhs) ops)))))
      (cond ((null (rest-of lhs)) lhs)
            (op (let ((rhs (parse-binary-r
                             (make-prog () (rest-of-rest lhs))
                             ops
                             parent-parser)))
                  (list (list (cdr op) (s-expr-of lhs) (s-expr-of rhs))
                        (rest-of rhs))))
            (t lhs)))))
```

The following functions implement expression parsing as described in the section discussing infix to prefix conversion. Each function implements a production on top of *parse-binary* or *parse-binary-r*. For instance, *parse-concat* parses concatenation operators and performs the following transformations:

```
a::b  ⟶  (cons a b)
a++b  ⟶  (append a b)
```

The factors (*a*,*b*) of these operations are recognized by *parse-factor*.

```
(define (parse-concat program)
  (parse-binary-r program
                  '((:: . cons)
                    (++ . append))
                  parse-factor))

; x^y  —>  (expt x y)

(define (parse-power program)
  (parse-binary-r program
                  '((^ . expt))
                  parse-concat))

; x*y   —>  (* x y)
; x/y   —>  (* x y)
; x//y  —>  (quotient x y)
; x\\y  —>  (remainder x y)

(define (parse-term program)
  (parse-binary program
                '((* . *)
                  (/ . /)
                  (// . quotient)
                  (\\ . remainder))
                parse-power))

; x+y  —>  (+ x y)
; x-y  —>  (- x y)

(define (parse-sum program)
  (parse-binary program
                '((+ . +)
                  (- . -))
                parse-term))

; x=y   —>  (= x y)
; x<>y  —>  ((lambda (x y) (not (= x y))) x y)
; x<y   —>  (< x y)
; x>y   —>  (> x y)
; x<=y  —>  (<= x y)
; x>=y  —>  (>= x y)

(define (parse-pred program)
  (parse-binary program
                '((= . =)
                  (<> . (lambda (x y) (not (= x y))))
                  (< . <)
                  (> . >)
                  (<= . <=)
                  (>= . >=))
                 parse-sum))
```

```
; x/\y  -->  (and x y)

(define (parse-conj program)
  (parse-binary program
                '((/\ . and))
                parse-pred))

; x\/y  -->  (or x y)

(define (parse-disj program)
  (parse-binary program
                '((\/ . or))
                parse-conj))
```

The *parse-expr* function is the front end of the expression parser. It converts a token list representing an M-expression into a program structure.

```
(define (parse-expr tlist)
  (parse-disj (make-prog () tlist)))
```

Internal-definition parses definitions that are part of `where` clauses. It returns a list containig the name and the body of the function being defined:

```
f[a1 ...] := expr  -->  (f (lambda (a1 ...) expr))
```

The resulting list has the form of a local definition as used by **`letrec`**.

```
(define (internal-definition program)
  (let ((head (parse-expr (rest-of program))))
    (cond ((eq (first-of-rest head) ':=)
            (let ((term (parse-expr (rest-of-rest head))))
              (make-prog
                (list (car (s-expr-of head))
                      (make-lambda
                            (cdr (s-expr-of head))
                            (s-expr-of term)))
                (rest-of term))))
          (t (bottom ':= 'expected 'at (rest-of program))))))
```

The *parse-compound* function parses the `where` part of a compound definition:

```
where f[x] := expr   -->  ((f (lambda (x) expr))
  and g[x] := expr         (g (lambda (x) expr))
  ...                      ...)
```

The resulting list can be used as an environment in **`letrec`**.

```
(define (parse-compound program)
  (letrec
    ((compound
       (lambda (prog def-list)
         (let ((defn (internal-definition (make-prog () prog))))
           (cond ((eq (first-of-rest defn) 'and)
                  (compound (rest-of-rest defn)
                            (cons (s-expr-of defn) def-list)))
                 (t (make-prog
                      (reverse (cons (s-expr-of defn) def-list))
                      (rest-of defn)))))))
    (compound program ())))
```

Create a **letrec** expression ouf of of an environment and a body:

```
(define (make-letrec env term)
  (list 'letrec env term))
```

Parse-definition parses simple definitions as well as compund definitions. It returns applications of **define**:

```
f[x] := expr        —>  (define (f x) expr)
f[x] := y           —>  (define (f x)
  where g[x] := z         (letrec ((g (lambda (x) z)))
                            y))
```

When *parse-definition* is called, the head of the definition already has been parsed, so *program* contains a partial program like this: `((f x) (:= expr))`. The resons will be explained below.

```
(define (parse-definition program)
  (let ((term (parse-expr (rest-of-rest program))))
    (cond ((eq (first-of-rest term) 'where)
            (let ((compound (parse-compound (rest-of-rest term))))
              (make-prog
                (list 'define
                      (s-expr-of program)
                      (make-letrec (s-expr-of compound)
                                   (s-expr-of term)))
                (rest-of compound))))
          (t (make-prog (list 'define
                              (s-expr-of program)
                              (s-expr-of term))
                        (rest-of term))))))
```

Parse-program parses a full M-expression program. Each program is either an expression or a definition, which introduces a problem: both kinds of sentence may share a common prefix of indefinite length:

```
f [ a1 ... a∞ ]
f [ a1 ... a∞ ] := expr
```

So the parser would need infinite look-ahead to decide what kind of input is contained in the token stream.

The problem is solved by *assuming* that the input contains an expression and parsing that. After parsing an expression the next input token is either a definition operator (`:=`) or not. When it is a definition operator, the partially translated program is passed to *parse-defintion*, which uses the S-expression generated so far as the head of a defintion.

So the ambiguity in the grammar is actually handled at the semantic level (by rewriting the meaning of an S-expression) rather than at the syntactic level. This is a common technique in hand-crafted compilers.

```
(define (parse-program tlist)
  (let ((program (parse-expr tlist)))
    (cond ((eq (first-of-rest program) ':=)
            (parse-definition program))
          (t program))))
```

M-expr-compile compiles an M-expression to an S-expression and returns it:

```
(mexpr-compile '(f[x] := x))  =>  (define (f x) x))
```

The ‘‘rest’’ part of the program structure returned by *parse-program* must be empty or a syntax error has occurred.

```
(define (mexpr-compile source)
  (let ((program (parse-program (tokenize source))))
    (cond ((end-of program)
            (car program))
          (t (bottom 'syntax 'error 'at (rest-of program))))))
```

M-expr-eval compiles and evaluates an M-expresion:

```
(define (mexpr-eval source)
  (eval (mexpr-compile source)))
```

The version of MEXPRC discussed in this chapter reads and compiles M-expressions, but emits results as S-expressions. Can you write a front end to MEXPRC that translates its output back to M-expression form? E.g.:

```
(mexpr '(1/2 ^ 1/2))       =>  '(1/4)
(mexpr '(%a :: <<b,c>>))  =>  '(<< a, b, c >>)
...
```

The hack that MEXPRC uses to distinguish between function applications and function definitions introduces a bug. It allows non-symbols in the formal argument lists of functions:

```
(mexpr-compile '(f[a+b, 17] := expr)) => '(define (f #+ab '#17) expr)
```

How would you fix this bug?

10.3 example programs

M-expression (MEXPR) programs can be stored in files, just like zenlisp programs. MEXPR programs must require MEXPRC, and the source code should be placed in the file as an argument of *mexpr-eval*, as the following example illustrates:

```
(require 'mexprc)
(mexpr-eval '(

  m_fac[x] := [x=0 -> 1: m_fac[x-1] * x]

))
```

MEXPR programs of this form can be loaded using **`load`**. The application of *mexpr-eval* compiles the code automatically when the file is loaded. So the above example can be used like this (given that the code is stored in the file `m_fac.l`):

```
; user input is in italics
(load m_fac)
=> :t
(mexpr-eval '(m_fac[17]))
=> '#355687428096000
```

10.3.1 append lists

This is an MEXPR implementation of the *append2* function introduced on page 22:

```
(require 'mexprc)

(mexpr-eval '(

  m_append[a,b] :=
    r_append[reverse[a], b]
    where
      r_append[a,b] :=
        [null[a]
           -> b:
         r_append[cdr[a], car[a]::b]]

))
```

Unfortunately, this function does not accept variable numbers of arguments. In the case of **`append`** this is not too bad, because

```
(append a b c d)
```

translates to

```
a ++ b ++ c ++ d
```

but variadic functions may be useful under other circumstances. Can you devise a notation for variadic MEXPR functions? Can you implement it by modifying MEXPRC?

10.3.2 the towers of hanoi

You *do* know the rules of the *Towers of Hanoi*, don't you? If not, here is a brief description:

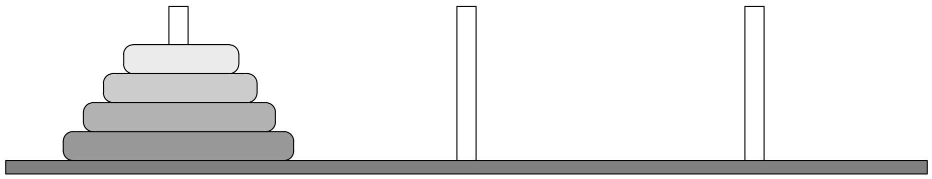
Fig. 8 – the towers of hanoi

There are three poles and a number of disks of different diameters as depicted in figure 8. Originally all disks are stacked on one pole in such a way that a smaller disk is always placed on top of a larger disk. The task is to move all disks to another pole — *but*:

– only one disk may be moved at a time;
– disks may only move from one pole to another;
– no larger disk may be placed on top of a smaller one.

Good luck!

Of course, being a programmer, you would not want to solve this puzzle in your head. You would want to find a *general* solution in the form of a program that finds the minimum set of moves required to solve the puzzle for a given number of disks.

Feel free to find such a solution before looking at the following MEXPR program. The solution is not too hard once you have understood how the puzzle is solved.

```
(require 'mexprc)

(mexpr-eval '(

  m_hanoi[n] :=
    solve[%LEFT, %MIDDLE, %RIGHT, n]
    where
      solve[from, to, via, n] :=
        [n=0
           -> nil:
         solve[from, via, to, n-1]
           ++ list[ list[from, to] ]
           ++ solve[via, to, from, n-1]]

))
```

What complexity does the *m_hanoi* program have? Can it be improved?

10.3.3 n queens

The rules of the *eight queens* puzzle are even simpler than those of the Towers of Hanoi: Place eight queens on a chess board in such a way that no queen may attack another. In case you do not play chess or prefer a less violent description: place eight objects in an 8×8 grid in such a way that no two objects can be connected by a horizontal, vertical, or diagonal line.

N queens is a generalization of this puzzle that uses an n×n grid instead of a chess board. The program of this section returns the first solution for a board of a given size, e.g.:

```
(mexpr-eval '(m_queens[4]))  =>  '(#1 #7 #8 #14)
```

Solutions are lists of fields. Fields are enumerated rather than giving them x/y coordinates. Above solution translates to the following grid (objects are placed on boldface numbers):

3 **7** 11 15

2 6 10 **14**

1 5 9 13

0 4 **8** 12

The problem is typically solved using *backtracking*. A piece is placed in the first row of each column and when it conflicts with a previously placed piece, the next row is tried. When there are no more rows in a column, the algorithm *backtracks* to the previous column and moves the piece in that column to the next row. The program finishes when a piece can be successfully placed in the last column or when it attempts to backtrack in the first column.

When a piece was placed in the last column, the algorithm found a solution. When it attempts to backtrack in the first column, no solution exists for the given grid size.

Obviously there is no solution for a 2×2 grid:

1 3   1 **3**   **1** 3   **1 3**
0 2   **0** 2   0 **2**   0 2

In the same way you can show that there is no solution for a 3×3 grid, but there already are 27 possible configurations to test. Larger grids are best checked by a program. Here it is:

```
(require 'mexprc)

(mexpr-eval '(

  m_queens[size] :=
    n_queens[0, 0, nil]

    where n_queens[q, c, b] :=
      [c = size
         -> reverse[b]:
```

```
        column[q] <> c
          -> [null[b]
                -> nil:
              n_queens[car[b]+1, c-1, cdr[b]]]:
        safe_place[q, b]
          -> n_queens[next_column[q], c+1, q::b]:
        n_queens[q+1, c, b]]

    and column[x] := x // size

    and row[x] := x \\ size

    and safe_place[x,b] :=
      [null[b]
         -> true:
       connected[car[b], x]
         -> false:
       safe_place[x, cdr[b]]]

    and connected[x,y] :=
      common_h_v[x,y] \/ common_dia[x,y]

    and common_h_v[x,y] :=
      row[x] = row[y] \/ column[x] = column[y]

    and common_dia[x,y] :=
      abs[column[x]-column[y]] = abs[row[x]-row[y]]

    and next_column[q] := [q+size] // size * size

))
```

This program prints only the first solution for a given grid size. Can you modify it to print all solutions?

What worst-case complexity does *m_queens* have? What average complexity does it have?

Do you think that solutions exist for all grid sizes greater than 3×3?

11. another micro kanren

The amk (*Another Micro Kanren*) package embeds declarative logic programming in zenlisp. It is based on ideas presented in ''The Reasoned Schemer'' [20] by Daniel P. Friedman, et al. The code is also inspired by the ''Sokuza Mini-Kanren'' implementation by Oleg Kiselyov.

Amk may be considered a language of its own. It is a logic programming language — like PROLOG — rather than a functional progamming language. However, it integrates seemlessly into zenlisp: zenlisp data may be passed to amk and amk returns ordinary S-expressions as its results.

In order to use amk in zenlisp programs, the **amk** package must be loaded by beginning a program with the following expression:

```
(require '~amk)
```

11.1 introduction

11.1.1 functions versus goals

In functional programming, functions are combined to form programs. For example, the **append** function of zenlisp concatenates lists:

```
(append '#orange '#-juice) => '#orange-juice
```

The basic building stones of logic programming are called *goals*. A goal is a function that maps knowledge to knowledge:

```
(run* () (appendo '#orange '#-juice '#orange-juice)) => '(())
```

An application of **run*** is called a *query*. **Run*** is the interface between zenlisp and amk. The result of **run*** is called the *answer* to the corresponding query.

The goal used in the above query is **append^0** [page 177].

An answer of the form `'(())` means ''success''. In the above example, this means that `'#orange-juice` is indeed equal to the concatenation of `'#orange` and `'#-juice`.

A goal returning a positive answer is said to *succeed*.

```
(run* () (appendo '#orange '#-juice '#fruit-salad)) => ()
```

does *not* succeed, because `'#fruit-salad` cannot be constructed by appending `'#orange` to `'#-juice`.

A goal that does not succeed is said to *fail*. Failure is represented by **()**.

20 Friedmann, Byrd, Kiselyov; "The Reasoned Schemer"; MIT Press, 2005

When one or more arguments of a goal are replaced with variables, the goal attempts to *infer* the values of these variables.

Logic variables are created by the **var** function:

```
(define vq (var 'q))
```

Any argument of a goal can be a variable:

```
(run* vq (appendo '#orange '#-juice vq)) => '(#orange-juice)
(run* vq (appendo '#orange vq '#orange-juice)) => '(#-juice)
(run* vq (appendo vq '#-juice '#orange-juice)) => '(#orange)
```

In these sample queries, **run*** is told that we are interested in the value of *vq*. It runs the given goal and then returns the value or values of *vq* rather than just success or failure.

Goals are non-deterministic, so a query may return more than a single answer:

```
(run* vq (let ((dont-care (var 'dont-care)))
           (appendo dont-care vq '#abcd)))
=> '(#abcd #bcd #cd #d ())
```

This query returns all values which give `'#abcd` when appended to something that is not interesting. In other words, it returns all suffixes of `'#abcd`.

Subsequently, the following query returns all prefixes:

```
(run* vq (let ((dont-care (var 'dont-care)))
           (appendo vq dont-care '#abcd)))
=> '(() #a #ab #abc #abcd)
```

What do you think is the answer to the following query?

```
(run* vq (let ((x (var 'x))
               (y (var 'y)))
           (appendo x y vq)))
```

Does it have an answer at all?

Answer: The query has no answer, because there is an indefinite number of combinations that can be used to form a concatenation with an unspecific prefix and suffix. So **append**^{0} never stops generating combinations of values for its variables.

11.1.2 unification

Unification is an algorithm that forms the heart of every logic programming system. The "unify" goal is written **==**. The query

```
(run* vq (== x y))
```

means "unify *x* with *y*". The answer of this query depends on the values of *x* and *y*:

```
(run* vq (== 'pizza 'pizza))  => '(())
(run* vq (== 'cheese 'pizza)) => ()
(run* vq (== vq vq))          => '(())
(run* vq (== 'cheese vq))     => '(cheese)
```

When two atoms are passed to **==**, it succeeds if the atoms are equal.

When a variable is passed to **==**, the variable is *bound* to the other argument:

```
(run* vq (== vq 'cheese)) => '(cheese)
(run* vq (== 'cheese vq)) => '(cheese)
```

The order of arguments does not matter.

When two variables are unified, these two variables are guaranteed to *always* bind to the same value:

```
(run* vq (let ((vx (var 'x)))
           (== vq vx)))
```

makes *vq* and *vx* bind to the same value. Binding a value to one of them at a later time automatically binds that value to both of them.

Non-atomic arguments are unified recursively by first unifying their car parts and then unifying their cdr parts:

```
(run* vq (== '(x (y) z) '(x (y) z))) => '(())
(run* vq (== '(x (y) z) '(x (X) z))) => ()
(run* vq (== vq '(x (y) z)))         => '((x #y z))
```

Inference works even if variables are buried inside of lists:

```
(run* vq (== (list 'x vq 'z)  '(x #y z))) => '(#y)
```

Because `'#y` is the only value for *vq* that makes the goal succeed, that value is bound to *vq*.

How does this work?

– `'x` is unified with `'x`;
– *vq* is unified with `'#y` (binding *vq* to `'#y`);
– `'z` is unified with `'z`.

Each unification may expand the "knowledge" of the system by binding a variable to a value or unifying two variables.

When unifying lists, the cdr parts of the lists are unified in the context of the knowledge gained during the unification of their car parts:

```
(run* vq (== '(     pizza fruit-salad)
              (list vq    vq         ))) => ()
```

This unification cannot succeed, because *vq* is first bound to `'pizza` and then the same variable is unified with `'fruit-salad`.

When *vq* is unified with `'pizza`, *vq* is *fresh.* A variable is fresh if it is not (yet) bound to any value.

Only fresh variables can be bound to values.

When a form is unified with a bound variable, it is unified with the *value* of that variable. Hence

```
(run* vq (== '(pizza fruit-salad) (list vq vq))) => ()
```

is equivalent to

```
(run* vq (== '(pizza fruit-salad) (list vq 'pizza))) => ()
```

The following query succeeds because no contradiction is introduced:

```
(run* vq (== '(pizza pizza) (list vq vq))) => '(pizza)
```

First *vq* is unified with `'pizza` and then *the value of vq* (which is `'pizza` at this point) is unified with `'pizza`.

Bound variables can still be unified with fresh variables:

```
(run* vq (let ((vx (var 'x)))
           (== (list 'pizza vq)
               (list  vx    vx)))) => '(pizza)
```

Here *vx* is unified with `'pizza` and then the fresh variable *vq* is unified with *vx*, binding *vq* and *vx* to the same value.

Again, the order of unification does not matter:

```
(run* vq (let ((vx (var 'x)))
           (== (list vq 'pizza)
               (list vx  vx)))) => '(pizza)
```

11.1.3 logic operators

The **`any`** goal succeeds, if at least one of its *subgoals* succeeds:

```
(run* vq (any (== vq 'pizza)
              (== 'orange 'juice)
              (== 'yes 'no)))
=> '(pizza)
```

In this example, one of the three subgoals succeeds and contributes to the anwer.

Because **`any`** succeeds if at least one of its subgoals succeeds, it fails if no subgoals are given:

```
(run* () (any)) => ()
```

Multiple subgoals of **any** may unify the same variable with different forms, giving a non-determistic answer:

```
(run* vq (any (== vq 'apple)
              (== vq 'orange)
              (== vq 'banana)))
=> '(apple orange banana)
```

No contradiction is introduced. *Vq* is bound to each of the three values.

The **any** goal implements the *union* of the knowledge gained by running its subgoals:

```
(run* vq (any fail
              (== vq 'fruit-salad)
              fail))
=> '(fruit-salad)
```

It succeeds even if some of its subgoals fail. Therefore it is equivalent to the *logical or*.

Fail is a goal that always fails.

The **all** goal is a cousin of **any** that implements the *logical and*:

```
(run* vq (all (== vq 'apple)
              (== 'orange 'orange)
              succeed))
=> '(apple)
```

Succeed is a goal that always succeeds.

All succeeds only if all of its subgoals succeed, but it does more than this.

All forms the intersection of the knowledge gathered by running its subgoals by removing any contradictions from their answers:

```
(run* vq (all (== vq 'apple)
              (== vq 'orange)
              (== vq 'banana))) => ()
```

This query fails because *vq* cannot be bound to `'apple` and `'orange` and `'banana` at the same time.

The effect of **all** is best illustrated in combination with **any**:

```
(run* vq (all (any (== vq 'orange)
                   (== vq 'pizza))
              (any (== vq 'apple)
                   (== vq 'orange))))
=> '(orange)
```

The first **any** binds *vq* to `'orange` or `'pizza` and the second one binds it to `'apple` or `'orange`.

All forms the intersection of this knowledge by removing the contradictions *vq*=`'pizza` and *vq*=`'apple`. *Vq*=`'orange` is no contradiction because it occurs in both subgoals of **all**.

BTW: **all** fails if at least one of its subgoals fails. Therefore it succeeds if no goals are passed to it:

```
(run* () (all)) => '(())
```

11.1.4 parameterized goals

A parameterized goal is a function returning a goal:

```
(define (conso a d p) (== (cons a d) p))
```

Applications of **conso** evaluate to a goal, so **cons**^{0} can be used to form goals in queries:

```
(run* vq (conso 'heads 'tails vq)) => '((heads . tails))
```

In the prose, **conso** is written **cons**^{0}. The trailing “o” of goal names is pronounced separately (e.g. “cons-oh”).

Obviously, **cons**^{0} implements something that is similar to the **cons** function.

However, **cons**^{0} can do more:

```
(run* vq (conso 'heads vq '(heads . tails))) => '(tails)
(run* vq (conso vq 'tails '(heads . tails))) => '(heads)
```

So **conso**^{0} can be used to define two other useful goals:

```
(define (caro p a) (conso a (_) p))
```

Car^0 is similar to the **car** function of zenlisp and **cdr**^{0} is similar to its **cdr** function:

```
(define (cdro p d) (conso (_) d p))
```

Like the _ variable of PROLOG, the expression **(_)** indicates a value that is of no interest.

When the second argument of **car**^{0} and **cdr**^{0} is a variable, they resemble **car** and **cdr**:

```
(run* vq (caro '(x . y) vq)) => '(x)
(run* vq (cdro '(x . y) vq)) => '(y)
```

Like **cons**^{0}, **car**^{0} and **cdr**^{0} can do more than their zenlisp counterparts, though:

```
(run* vq (caro vq 'x)) => '((x . _,0))
(run* vq (cdro vq 'y)) => '((_,0 . y))
```

The query

```
(run* vq (caro vq 'x))
```

asks: “what has a car part of `'x`?” and the answer is “any pair that has a car part of `'x` and a cdr part that does not matter.”

Clever, isn’t it?

11.1.5 reification

Atoms of the form `_,n`, where *n* is a unique number, occur whenever an answer would otherwise contain fresh variables:

```
(run* vq (let ((vx (var 'x))
               (vy (var 'y))
               (vz (var 'z)))
           (== vq (list vx vy vz))))
=> '((_,0 _,1 _,2))
```

In the remainder of this document, `_,n` may be spelled $\_{n}$.

$\_{0}$, $\_{1}$, etc are called *reified variables.*

The replacement of fresh variables with reified names is called *reification.* It replaces each fresh variable with a unique “item” (*res* being the latin word for “item”).

11.1.6 recursion

Here is a recursive zenlisp predicate:

```
(define (mem-p x l)
  (cond ((null l) :f)
        ((eq x (car l)) :t)
        (t (mem-p x (cdr l)))))
```

`Mem-p` tests whether *l* contains *x*:

```
(mem-p 'c '#abcdef) => :t
(mem-p 'x '#abcdef) => :f
```

In amk you cannot write code like **`(eq x (car l))`** because

In logic programming, there is no function composition.

Each argument of a goal *must* be either a datum or a variable. Only **`any`** and **`all`** have subgoals:

```
(define (memo x l)
  (let ((va (var 'a))
        (vd (var 'd)))
    (any (all (caro l va)
              (eqo x va))
         (all (cdro l vd)
              (lambda (s)
                ((memo x vd) s))))))
```

Here are some observations:

– One **any** containing one or multiple **all** goals is the logic programming equivalent of **cond**;
– Each time **mem^0** is entered, a fresh *va* and *vd* is created;
– **Mem^0** does not seem to check whether *l* is **()**;
– The recursive case uses "eta conversion" to avoid early recursion (see below).

Does **mem^0** work? Yes:

```
(run* () (memo 'c '#abcdef)) => '(())
(run* () (memo 'x '#abcdef)) => ()
```

How does it work?

The first **all** unifies the car part of *l* with *va*. In case *l*=**()**, **all** fails.

Eq^0 is a synonym for **==**.

If *va* (which is now an alias of `(car l)`) can be unified with *x*, this branch of **any** succeeds.

If *l* is **()**, both of these goals fail:

```
(caro l va) => ()
(cdro l vd) => ()
```

and so the entire **mem^0** fails. There is no need to test for *l*=**()** explicitly.

The second **all** of **mem^0** unifies *vd* with the cdr part of *l* and then recurses.

Because **any** and **all** are ordinary zenlisp functions, recursion would occur *before* running **all**, if the application of **mem^0** was not protected by eta expansion.

Eta expansion wraps a lambda function around another function (and eta reduction removes that "wrapper"):

```
(memo x vd)  <------->  (lambda (s) ((memo x vd) s))
                eta
```

Both conversions maintain the meaning of the original function, except for the time of its reduction: **(memo x vd)** would be reduced immediately (delivering a goal) while in

```
(lambda (s) ((memo x vd) s))
```

the goal is only created when the enclosing lambda function is applied to a value. Eta expansion effectively implements "*call-by-name*" semantics.

All recursive cases must be protected using eta expansion.

11.1.7 converting predicates to goals

A predicate is a function returning a truth value.

Each goal is a predicate in the sense that it either fails or succeeds.

There are five steps involved in the conversion of a predicate to a goal:

`c1.` Decompose function compositions;
`c2.` Replace functions by parameterized goals;
`c3.` Replace **`cond`** with **`any`** and its clauses with **`all`**;
`c4.` Remove subgoals that make the predicate fail;
`c5.` Protect recursive cases using eta expansion.

`Mem-p` [page 171] is converted as follows:

```
((eq x (car l)) :t)
```

becomes (by c1, c2, c3)

```
(let ((va (var 'a)))
  (all (caro l va)
       (eqo x va)))
```

and

```
(t (mem-p x (cdr l)))
```

becomes (by c1, c2, c3, c5)

```
(let ((vd (var 'd)))
  (all (cdro l vd)
       (lambda (s)
         ((memo x vd) s))))
```

Finally

```
((null l) :f)
```

is removed (by c4) and **`cond`** is replaced with **`any`** (by c3).

In the original definition of **`mem`**^{0}, `(let ((va ...)))` and `(let ((vd ...)))` are combined and moved outside of **`any`**.

BTW, the application of **`eq`**^{0} in **`mem`**^{0} is redundant, because **`car`**^{0} itself could make sure that *x* is the car part of *l*. So **`mem`**^{0} can be simplified significantly:

```
(define (memo x l)
  (let ((vt (var 't)))
    (any (caro l x)
         (all (cdro l vt)
              (lambda (s)
                ((memo x vt) s))))))
```

11.1.8 converting functions to goals

Memq is similar to **mem-p**:

```
(define (memq x l)
  (cond ((null l) :f)
        ((eq x (car l)) l)
        (t (memq x (cdr l)))))
```

Instead of returning just **:t** in case of success, it returns the first sublist of *l* whose car part is *x*:

```
(memq 'orange '(apple orange banana)) => '(orange banana)
```

Functions are converted to goals in the same way as predicates, but there is one additional rule:

`c6.` Add an additional argument to unify with the result.

Memq^0 is similar to **mem^0**, but it has an additonal argument *r* for the result and an additional goal which unifies the result with *r*:

```
(define (memqo x l r)
  (let ((vt (var 't)))
    (any (all (caro l x)
              (== l r))
         (all (cdro l vt)
              (lambda (s)
                ((memqo x vt r) s))))))
```

Like **memq**, **memq^0** can be queried to deliver the first sublist of *l* whose head is *x*:

```
(run* vq (memqo 'orange '(apple orange banana) vq))
=> '((orange banana))
```

Memq^0 even delivers *all* the sublists of *l* beginning with *x*:

```
(run* vq (memqo 'b '#abababc vq))
=> '(#bababc #babc #bc)
```

If you are only interested in the first one, take the car part of the anwer.

Memq^0 can be used to implement the identity function:

```
(run* vq (memqo vq '(orange juice) (_)))
=> '(orange juice)
```

How does this work?

The question asked here is “what should *vq* be unified with to make `(memqo vq '(orange juice) (_))` succeed?”

The **car^0** in **memq^0** unifies *vq* (which is unified with *x*) with `'orange` and because *vq* is fresh, it succeeds.

The second case also succeeds. It binds *l* to `'(juice)` and retries the goal. In this branch, *vq* is still fresh.

The **car^0** in **memq^0** unifies *vq* with `'juice` and because *vq* is fresh, it succeeds.

The second case also succeeds. It binds *l* to `()` and retries the goal. In this branch, *vq* is still fresh.

`(memqo vq () (_))` fails, because neither **car^0** nor **cdr^0** can succeed with *l*=**`()`**.

Any forms the union of *vq*=`'orange` and *vq*=`'juice`, which is the answer to the query.

11.1.9 cond versus any

LISP's **cond** pseudo function tests the predicates of its clauses sequentially and returns the normal form of the expression associated with the first true predicate:

```
(cond (t  'bread)
      (:f 'with)
      (t  'butter)) => 'bread
```

Even though the clause `(t 'butter)` also has a true predicate, the above **cond** will *never* return `'butter`.

A combination of **any** and **all** can be used to form a logic programming equivalent of **cond**:

```
(run* vq (any (all succeed (== vq 'bread))
              (all fail    (== vq 'with))
              (all succeed (== vq 'butter))))
=> '(bread butter)
```

Any replaces **cond** and **all** introduces each individual case.

Unlike **cond**, though, this construct returns the values of *all* cases that succeed.

While **cond** ignores the remaining clauses in case of success, **any** keeps trying until it runs out of subgoals.

This is the reason why **memq^0** [page 174] returns all sublists starting with a given form:

```
(run* vq (memqo 'b '#abababc vq)) => '(#bababc #babc #bc)
```

This is how it works:

When the head of *l* is *not* equal to `'b`, the first subgoal of **any** in **memq^0** fails, so nothing is added to the answer.

When the head of *l* is equal to `'b`, the first subgoal of **any** succeeds, so *l* is added to the answer.

In either case, the second goal is tried. It succeeds as long as *l* can be decomposed. It fails when the end of the list *l* has been reached.

When the second goal succeeds, the whole **any** is tried on the cdr part of *l*, which may add more sublists to the answer.

What happens when the order of cases is reversed in memq^0?

```
(define (rmemqo x l r)
  (let ((vt (var 't)))
    (any (all (cdro l vt)
              (lambda (s)
                ((rmemqo x vt r) s)))
         (all (caro l x)
              (== l r)))))
```

Because **any** keeps trying until it runs out of goals, **rmemq**^{0} does indeed return all matching sublists, just like **memq**^{0}. However...

```
 (run* vq (memqo 'b '#abababc vq)) => '(#bababc #babc #bc)
(run* vq (rmemqo 'b '#abababc vq)) => '(#bc #babc #bababc)
```

Because **rmemq**^{0} first recurses and then checks for a matching sublist, its answer lists the last matching sublist first.

Reversing the goals of **memq**^{0} makes it return its results in reverse order.

While **memq**^{0} can implement the identity function, **rmemq**^{0} can implement a function that reverses a list:

```
 (run* vq (memqo vq '(ice water) (_))) => '(ice water)
(run* vq (rmemqo vq '(ice water) (_))) => '(water ice)
```

11.1.10 first class variables

Logic variables are first class values.

When a bound logic variable is used as an argument of a goal, the value of that variable is passed to the goal:

```
(run* vq (let ((vx (var 'vx)))
           (all (== vx 'piece-of-cake)
                (== vq vx))))
=> '(piece-of-cake)
```

When a fresh variable is used as an argument of a goal, the *variable itself* is passed to that goal:

```
(run* vq (let ((vx (var 'x)))
           (== vq vx)))
=> '(_,0)
```

Because the variable *vx* is fresh, it is reified by the interpreter after running the query, giving $\_{0}$.

Variables can even be part of compound data structures:

```
(run* vq (let ((vx (var 'x)))
           (conso 'heads vx vq)))
=> '((heads . _,0))
```

Unifying a variable that is part of a data structure at a later time causes the variable part of the data structure to be “filled in” belatedly:

```
(run* vq (let ((vx (var 'x)))
           (all (conso 'heads vx vq)
                (== vx 'tails))))
=> '((heads . tails))
```

The **append^0** goal makes use of this fact:

```
(define (appendo x y r)
  (any (all (== x ())
            (== y r))
       (let ((vh  (var 'h))
             (vt  (var 't))
             (vtr (var 'tr)))
         (all (conso vh vt  x)
              (conso vh vtr r)
              (lambda (s)
                ((appendo vt y vtr) s))))))
```

How is the following query processed?

```
(run* vq (appendo '#ab '#cd vq))
```

In its recursive case, **append^0** first decomposes *x*=`'#ab` into its head *vh*=`'a` and tail *vt*=`'#b`:

```
(conso vh vt x)
```

The next subgoal states that the head *vh* consed to *vtr* (the **t**ail of the **r**esult) gives the result of **append^0**:

```
(conso vh vtr r)
```

Because *vtr* is fresh at this point, *r* is bound to a structure containing a variable:

r_0 = (cons 'a vtr_0)

Vtr and *r* are called vtr_0 and r_0 here, because they are the first instances of these variables.

When the goal recurses, vtr_0 is passed to **append^0** in the place of *r*:

(appendo '#b '#cd vtr_0)

Append^0 creates fresh instances of *vtr* and *r* (called vtr_1 and r_1).

At this point r_1 and vtr_0 may be considered *the same variable*, so

(conso vh vtr_1 r_1)

results in

r_1 = vtr_0 = (cons 'b vtr_1)

and

r_0 = (cons 'a vtr_0) = (cons 'a (cons 'b vtr_1))

When **append^0** recurses one last time, vtr_1 is passed to the goal in the place of r and the instance r_2 is created:

(appendo () '#cd vtr_1)

Because x=`()`, the subgoal handling the trivial case is run:

(== y r_2)

And because r_2 and vtr_1 are the same variable,

r_2 = vtr_1 = '#cd
r_1 = vtr_0 = (cons 'b vtr_1)
            = (cons 'b '#cd)
r_0 = (cons 'a vtr_0)
   = (cons 'a (cons 'b vtr_1))
   = (cons 'a (cons 'b '#cd))

11.1.11 first class goals

Like LISP functions, goals are first class values.

The *filter^0* goal makes use of this fact:

```
(define (filtero p l r)
  (let ((va (var 'a))
        (vd (var 'd)))
    (any (all (caro l va)
              (p va)
              (== va r))
         (all (cdro l vd)
              (lambda (s)
                ((filtero p vd r) s))))))
```

Filter^0 extracts all members with a given property from a list.

The property is described by the goal p which is passed as an argument to *filter^0*:

```
(run* vq (filtero pairo '(a b (c . d) e (f . g)) vq))
=> '((c . d) (f . g))
```

Pair^0 is defined this way:

```
(define (pairo x) (conso (_) (_) x))
```

Because parameterized goals are ordinary functions, though, there is no need to invent a new function name. **Lambda** works fine:

```
(run* vq (filtero (lambda (x) (conso (_) (_) x))
                  '(a b (c . d) e (f . g)) vq))
=> '((c . d) (f . g))
```

11.1.12 negation

The **neg** goal succeeds if its subgoal fails, and fails if its subgoal succeeds:

```
(run* () (neg fail)) => '(())
(run* () (neg succeed)) => ()
```

Neg *never* contributes any knowledge:

When its subgoal succeeds, **neg** itself fails, thereby deleting all knowledge gathered by its subgoal.

When its subgoal fails, there is no knowledge to add.

However, **neg** is not as straight-forward as it seems to be:

```
(define (nullo x) (eqo () x))
(run* vq (neg (nullo vq)))
```

The **null**^{0} goal tests whether its argument is **()**.

What should be the answer to the question "what is not equal to **()**?"

Neg answers this question using an approach called the "closed world assumption", which says "what cannot be proven true must be false".

So the answer to above question is "nothing". Because the value of *vq* is not known, **neg** cannot prove that it is not equal to **()** and fails:

```
(run* vq (neg (nullo vq))) => ()
```

Technically, it works like this:

Vq is fresh, so **null**^{0} unifies it with **()** and succeeds. Because **null**^{0} succeeds, **neg** must fail.

This approach has its consequences:

```
(run* vq                               (run* vq
      (all (any (== vq 'orange)              (all (neg (== vq 'pizza))
                (== vq 'pizza)                    (any (== vq 'orange)
                (== vq 'ice-cream))                    (== vq 'pizza)
           (neg (== vq 'pizza))))                      (== vq 'ice-cream))))
=> '(orange ice-cream)                 => ()
```

Depending on its context, **neg** has different functions.

In the right example it makes the entire query fail, because the fresh variable *vq* can be unified with `'pizza`.

In the left example, where *vq* already has some values, it eliminates the unification of *vq* and `'pizza`.

Therefore

Negation should be used with great care.

11.1.13 cutting

The **memq^0** goal [page 174] returned all sublists whose heads matched a given form:

```
(run* vq (memqo 'b '#abababc vq)) => '(#bababc #babc #bc)
```

For the case that you are *really, really* only interested in the first match, there is a technique called *cutting*.

It is implemented by the **one** goal:

```
(run* vq (one fail
              (== vq 'apple)
              (== vq 'pie)))
=> '(apple)
```

As soon as one subgoal of **one** succeeds, **one** itself succeeds immediately and “cuts off” the remaining subgoals.

The name of **one** suggests that at most **one** of its subgoals can succeed.

Using **one**, a variant of **memq^0** can be implemented which succeeds with the first match:

```
(define (firsto x l r)
  (let ((vd (var 'd)))
    (one (all (caro l x)
              (== r l))
         (all (cdro l vd)
              (lambda (s)
                ((firsto x vd r) s))))))
```

The only difference between **memq^0** and *first^0* is that *first^0* uses **one** in the place of **any**.

First^0 cuts off the recursive case as soon as the first case succeeds:

```
(run* vq (firsto 'b '#abababc vq)) => '(#bababc)
```

So **one** is much more like **cond** than **any**.

However, **one** supresses *backtracking*, which is one of the most interesting properties of logic programming systems.

Here is another predicate:

```
(define (juiceo x)
  (let ((vt (var 't)))
    (all (cdro x vt)
         (caro vt 'juice))))
```

Juice^0 succeeds, if its argument is a list whose second element is equal to `'juice`, e.g.:

```
(run* () (juiceo '(orange juice))) => '(())
(run* () (juiceo '(cherry juice))) => '(())
(run* () (juiceo '(apply  pie  ))) => ()
```

Given the *juice^0* predicate, **memq^0** can be used to locate your favorite juice on a menu:

```
(define menu '(apple pie    orange pie    cherry pie
               apple juice  orange juice  cherry juice))
(run* vq (all (memqo 'orange menu vq)
              (juiceo vq)))
=> '((orange juice cherry juice))
```

When **memq^0** finds the sublist starting with the `'orange` right before `'pie`, *juice^0* fails and backtracking is initiated.

Memq^0 then locates the next occurrence of `'orange` and this time *juice^0* succeeds.

Using *first^0* supresses backtracking and so our favorite juice is never found:

```
(run* vq (all (firsto 'orange menu vq)
              (juiceo vq)))
=> ()
```

Therefore

Cutting should be used with great care.

11.3 a logic puzzle

The *Zebra Puzzle* is a well-known logic puzzle.

It is defined as follows:

– Five persons of different nationality live in five houses in a row.
– The houses are painted in different colors.
– The persons enjoy different drinks and brands of cigarettes.
– All persons own different pets.
– The Englishman lives in the red house.
– The Spaniard owns a dog.
– Coffee is drunk in the green house.
– The Ukrainian drinks tea.
– The green house is directly to the right of the ivory house.
– The Old Gold smoker owns snails.
– Kools are being smoked in the yellow house.
– Milk is drunk in the middle house.
– The Norwegian lives in the first house on the left.
– The Chesterfield smoker lives next to the fox owner.
– Kools are smoked in the house next to the house where the horse is kept.
– The Lucky Strike smoker drinks orange juice.
– The Japanese smokes Parliaments.
– The Norwegian lives next to the blue house.

Who owns the zebra?

To solve the puzzle, two questions have to be answered:

– How to represent the data?
– How to add facts?

Five attributes are linked to each house, so the row of houses can be represented by a list of 5-tuples like this:

(*nation cigarette drink pet color*)

Known facts are represented by symbols and unknown ones by variables.

The fact “the Spaniard owns a dog” would look like this:

```
(list 'spaniard (var 'cigarette) (var 'drink) 'dog (var 'color))
```

The addition of facts is explained by means of a simpler variant of the puzzle with only two attributes and two houses:

```
(list (list (var 'person) (var 'drink))
      (list (var 'person) (var 'drink)))
```

– In one house lives a Swede.
– In one house lives a tea drinker.
– In one house lives a Japanese who drinks coffee.
– The tea drinker lives in the left house.

Applying the first fact yields the following options (variables are rendered in italics):

```
'( ((Swede drink) house)
   (house (Swede drink)) )
```

This means that the Swede (whose drink is unknown) can live in the first or in the second house.

Adding the second fact yields more options:

```
'( ((Swede Tea)   house)
   ((Swede drink) (person Tea))
   ((person Tea)  (Swede drink))
   (house         (Swede Tea))   )
```

The key to the application of facts is unification. The fact

```
(list 'Swede (var 'drink))
```

can be unified with any fresh variable like *h* (as in *h*ouse):

```
(define h (var 'h))

(run* h (== h (list 'Swede (var 'drink))))
=> '((swede _,0))
```

To create *all* possible outcomes, the fact must be applied to *each* of the two houses:

```
(define fact (list 'Swede (var 'drink)))

(run* h (let ((h1 (var 'house1))
              (h2 (var 'house2)))
          (all (== h (list h1 h2))
               (any (== fact h1)
                    (== fact h2)))))
=> '(((swede _,0) _,1)
     (_,0 (swede _,1)))
```

Remember: reified variables like $\_{}_0$ and $\_{}_1$ denote something that is not known and/or of no interest.

In the above answer, $\_{}_0$ represents an unknown drink in the first outcome and an unknown house in the second one. $\_{}_1$ represents an unknown house in the first outcome and an unknown drink in the second one.

Each new fact must be unified with all outcomes produced so far.

A goal which automatically unifies a fact with all outcomes found so far would be helpful. The **mem^0** goal, which was defined earlier in this chapter [page 173], can do this.

Mem^0 tries to unify a given form with each member of a list. Replace "form" with "fact" and "list" with "outcome" and here we go:

```
(run* h (all (== h (list (var 'h1)
                         (var 'h2)))
             (memo (list 'Swede (var 'drink)) h)
             (memo (list (var 'person) 'Tea)  h)))
=> '(((swede tea) _,0)
     ((swede _,0) (_,1 tea))
     ((_,0 tea) (swede _,1))
     (_,0 (swede tea)))
```

At this point the query is *underspecified*; the known facts are not sufficient to tell where the Swede lives or whether he drinks tea or not.

By adding the third fact, some outcomes are eliminated:

```
(run* h (all (== h (list (var 'house1)
                         (var 'house2)))
             (memo (list 'Swede (var 'drink)) h)
             (memo (list (var 'person) 'Tea)  h)
             (memo (list 'Japanese 'Coffee)   h)))
=> '(((swede tea) (japanese coffee))
     ((japanese coffee) (swede tea)))
```

The query is still underspecified, but because the third fact contradicts the assumption that the other person drinks tea, we now know that the Swede drinks tea. We also know that the other person is a Japanese and drinks coffee.

To add the final fact, another goal is needed. *Left^0* checks whether *x* is directly on the left of *y* in the list *l*:

```
(define (lefto x y l)
  (let ((vt (var 't)))
    (any (all (caro l x)
              (cdro l vt)
              (caro vt y))
```

```
          (all (cdro l vt)
               (lambda (s)
                 ((lefto x y vt) s))))))
```

Using *left*^{0}, the puzzle can be solved:

```
(run* h (all (== h (list (var 'h1)
                         (var 'h2)))
             (memo  (list 'Swede (var 'drink)) h)
             (memo  (list (var 'person) 'Tea)  h)
             (memo  (list 'Japanese 'Coffee)   h)
             (lefto (list (var 'person) 'Tea)
                    (var 'house)               h)))
=> '(((swede tea) (japanese coffee)))
```

To solve the Zebra Puzzle, another predicate is needed. It expresses that *x* is *next to y*.

X is next to *y*, if *x* is on the left of *y* or *y* is on the left of *x*, so:

```
(define (nexto x y l)
  (any (lefto x y l)
       (lefto y x l)))
```

Predicates expressing the position of a house in the row are not required, because houses can be placed directly in the initial record:

```
(list (list 'norwegian (var 'c1) (var 'd1) (var 'p1) (var 'o1))
      (var 'h2)
      (list (var 'n2) (var 'c2) 'milk (var 'p2) (var 'o2))
      (var 'h4)
      (var 'h5))
```

Having to invent lots of unique variable names for unknown parameters is a bit cumbersome, but because the initial values of these variables are not really interesting, they can be replaced with anonymous variables:

```
(list (list 'norwegian (_) (_) (_) (_))
      (_)
      (list (_) (_) 'milk (_) (_))
      (_)
      (_))
```

Here is the full code for solving the Zebra Puzzle:

```
(define (zebra)
  (let ((h (var 'h)))
    (run* h (all (== h (list (list 'norwegian (_) (_) (_) (_))
                             (_)
                             (list (_) (_) 'milk (_) (_))
                             (_)
                             (_)))
                 (memo  (list 'englishman (_) (_) (_) 'red) h)
                 (lefto (list (_) (_) (_) (_) 'green)
                        (list (_) (_) (_) (_) 'ivory) h)
                 (nexto (list 'norwegian (_) (_) (_) (_))
                        (list (_) (_) (_) (_) 'blue) h)
                 (memo  (list (_) 'kools (_) (_) 'yellow) h)
                 (memo  (list 'spaniard (_) (_) 'dog (_)) h)
                 (memo  (list (_) (_) 'coffee (_) 'green) h)
                 (memo  (list 'ukrainian (_) 'tea (_) (_)) h)
                 (memo  (list (_) 'luckystrikes 'orangejuice (_) (_)) h)
                 (memo  (list 'japanese 'parliaments (_) (_) (_)) h)
                 (memo  (list (_) 'oldgolds (_) 'snails (_)) h)
                 (nexto (list (_) (_) (_) 'horse (_))
                        (list (_) 'kools (_) (_) (_)) h)
                 (nexto (list (_) (_) (_) 'fox (_))
                        (list (_) 'chesterfields (_) (_) (_)) h)
;                 (memo  (list (_) (_) 'water (_) (_)) h)
                 (memo  (list (_) (_) (_) 'zebra (_)) h)))))
```

The program should be run with a node pool size of 1024K bytes (i.e. start the zenlisp interpreter using the command: `zl -n 1024K`).

```
(zebra) => '(((norwegian kools _,0 fox yellow)
              (ukrainian chesterfields tea horse blue)
              (englishman oldgolds milk snails red)
              (japanese parliaments coffee zebra green)
              (spaniard luckystrikes orangejuice dog ivory)))
```

Note that the puzzle is in fact underspecified. The drink of the Norwegian is not known. In case you prefer a fully specified query, just uncomment the fact listing "water" as a drink in the above code.

11.3 implementation

The complete implementation is written in purely symbolic LISP.

The only library function that is needed is **`length`**. It is imported from zenlisp's `nmath` package.

```
(define amk :t)

(require '~nmath)
```

11.3.1 basics

These are the **`fail`** and **`succeed`** goals. They are guaranteed to have disjoint results:

```
(define (fail x) ())

(define (succeed x) (list x))
```

Var creates a logic variable and *var-p* checks whether an object is a logic variable. Logic variables are represented by forms like `(? . x)` where *x* is the name of the variable. Note that all logic variables are unique instances. [See page 55]

```
(define (var x) (cons '? x))

(define (var-p x)
  (and (not (atom x))
       (eq (car x) '?)))
```

Because all variables are unique, **var** can be used to create anonymous variables:

```
(define (_) (var '_))
```

Empty-s represents ignorance:

```
(define empty-s ())
```

Knowledge is represented by *substitutions*. Substitutions are implemented using association lists. *Empty-s* is an empty substitution.

Ext-s adds the association of the variable *x* with the value *v* to the substitution *s*.

```
(define (ext-s x v s) (cons (cons x v) s))
```

Walk looks up the value of *x* in the substitution *s*:

`(walk` *vx* `'((`*vx* `. bread))) => 'bread`

(*Vx* denotes the logic variable *x*, i.e. the form `(? . x)`.)

Walk may look like **assoc** or **assq**, but it does more:

```
(define (walk x s)
  (cond ((not (var-p x)) x)
        (t (let ((v (assq x s)))
             (cond (v (walk (cdr v) s))
                   (t x))))))
```

When the value associated with a variable is another variable, *walk* looks up that other variable, thereby following chains of variables:

`(walk` *vx* `'((`*vx* `.` *vy*`) (`*vz* `. sushi) (`*vy* `.` *vz*`))) => 'sushi`

This is how the unification of variables is implemented.

When the variable passed to *walk* is fresh or a fresh variable is found while following a chain of variables, the fresh variable is returned:

`(walk` *vx* `empty-s) =>` *vx*

This is why fresh variables are first-class objects.

The *atomic* predicate is used instead of *atom*, because zenlisp implements closures as lists.

```
(define (atomic x)
  (or (atom x)
      (eq (car x) 'closure)))
```

Unify is the heart of amk. It unifies *x* with *y*, looking up values of variables in *s*.

```
(define (unify x y s)
  (let ((x (walk x s))
        (y (walk y s)))
    (cond ((eq x y) s)
          ((var-p x) (ext-s x y s))
          ((var-p y) (ext-s y x s))
          ((or (atomic x) (atomic y)) :f)
          (t (let ((s (unify (car x) (car y) s)))
               (and s (unify (cdr x) (cdr y) s)))))))
```

Upon success *unify* returns *s* or an extension of *s* with new substitutions added. In case of failure it **:f**.

11.3.2 goals

This is the **==** goal. **==** is like *unify*, but it succeeds or fails rather than returning a substitution or falsity.

```
(define (== g1 g2)
  (lambda (s)
    (let ((s2 (unify g1 g2 s)))
      (cond (s2 (succeed s2))
            (t (fail s))))))
```

Note that **==** returns a lambda function that must be applied to a substitution to let the unification take place:

```
(== vq 'orange-juice) => {closure #s}
((== vq 'orange-juice) empty-s) => '(((vq . orange-juice)))
```

Also note that when **==** succeeds, it adds another list around the resulting substitution.

Here is the **any** goal:

```
(define (any . g*)
  (lambda (s)
    (letrec
      ((try (lambda g*
              (cond ((null g*) (fail s))
                    (t (append ((car g*) s)
                               (apply try (cdr g*))))))))
      (apply try g*))))
```

It forms a *list of substitutions* by applying each member of the list of goals *g** to the given knowledge *s* and appending the results:

```
((any (== vq 'ice) (== vq 'cream)) empty-s)
=> '(((vq . ice)) ((vq . cream)))
```

In the list of substitutions returned by **any**, each individual substitution is free of conflicting associations.

Any is the only goal that may produce multiple substitutions.

Here is **all**:

```
(define (all . g*)
  (lambda (s)
    (letrec
      ((try (lambda (g* s*)
              (cond ((null g*) s*)
                    (t (try (cdr g*)
                            (apply append
                                   (map (car g*) s*))))))))
      (try g* (succeed s)))))
```

All applies its subgoals to the knowledge *s**.

Because some of its subgoals may be **any**s (which may produce multiple substitutions), **all** maps each goal over a list of substitutions.

The mapping eliminates all associations from all substitutions of *s** that contradict the current goal. The remaining results are appended to form a new list of substitutions $s*_1$.

The next subgoal of **all** is mapped over $s*_1$, giving $s*_2$, etc.

Each subgoal is applied to the conjunction of the subgoals applied so far.

One is similar to **any**, but instead of appending substitutions, it returns the result of the first subgoal that succeeds:

```
(define failed null)

(define (one . g*)
  (lambda (s)
    (letrec
      ((try (lambda g*
              (cond ((null g*) (fail s))
                    (t (let ((out ((car g*) s)))
                         (cond ((failed out)
                                 (apply try (cdr g*)))
                               (t out))))))))
      (apply try g*))))
```

Here is the **`neg`** goal. Its implementation is more straight-forward than its application:

```
(define (neg g)
  (lambda (s)
    (let ((out (g s)))
      (cond ((failed out) (succeed s))
            (t (fail s))))))
```

`Choice` is a utility goal that facilitates the creation of sets:

```
(run* vq (choice vq '(orange banana kiwi)))
```

is an abbreviation of

```
(run* vq (any (== vq 'orange)
              (== vq 'banana)
              (== vq 'kiwi)))
```

Here is the code:

```
(define (choice x lst)
  (cond ((null lst) fail)
        (t (any (== x (car lst))
                (choice x (cdr lst))))))
```

11.3.2 interface

Occurs and *circular* are helper functions that will be used by *walk**, which is explained right after them.

Occurs checks whether the symbol or variable *x* occurs in the form *y*. Like *walk*, *occurs* follows chains of variables. Values of variables are looked up in *s*.

```
(define (occurs x y s)
  (let ((v (walk y s)))
    (cond ((var-p y) (eq x y))
          ((var-p v) (eq x v))
          ((atomic v) :f)
          (t (or (occurs x (car v) s)
                 (occurs x (cdr v) s))))))
```

A value of a variable that contains references to that variable is said to be *circular*:

```
((== vq (list vq)) empty-s) => '(((vq . (vq))))
```

A circular answer is not valid because it is self-referential.

The *circular* function checks whether the value of a variable is circular:

```
(define (circular x s)
  (let ((v (walk x s)))
    (cond ((eq x v) :f)
          (t (occurs x (walk x s) s)))))
```

*Walk** is like *walk*: it turns a variable *x* into a value *v*. In addition it replaces all variables found in *v* with their values.

*Walk** makes the answers computed by amk comprehensible:

```
((all (== vq (list vx)) (== vx 'foo)) empty-s)
  => '(((vx . foo) (vq . (vx))))
(walk* vq '((vx . foo) (vq . (vx))))
  => '(foo)
```

When *walk** encounters a fresh variable, it leaves it in the result.

When the variable to be *walk**ed is bound to a circular value, *walk** returns `:bottom`.

```
(define :bottom (var ':bottom))

(define (walk* x s)
  (letrec
    ((w* (lambda (x s)
           (let ((x (walk x s)))
             (cond ((var-p x) x)
                   ((atomic x) x)
                   (t (cons (w* (car x) s)
                            (w* (cdr x) s))))))))
    (cond ((circular x s) :bottom)
          ((eq x (walk x s)) empty-s)
          (t (w* x s)))))
```

Reify-name generates a reified name.

```
(define (reify-name n)
  (implode (append '#_, n)))
```

Reify creates a substitution in which each fresh variable contained in the form *v* is associated with a unique reified name:

```
(reify (list vx vy vz)) => '((vz . _,2) (vy . _,1) (vx . _,0))
```

The value *v* that is passed to *reify* must have been *walk**ed before.

```
(define (reify v)
  (letrec
    ((reify-s
       (lambda (v s)
         (let ((v (walk v s)))
           (cond ((var-p v)
                   (ext-s v (reify-name (length s)) s))
                 ((atomic v) s)
                 (t (reify-s (cdr v)
                             (reify-s (car v)
                                      s))))))))
    (reify-s v empty-s)))
```

Preserve-bottom implements bottom preservation. No surprise here.

Explanation: The term *bottom* is used in mathematics to denote an undefined result, like a diverging function. *Bottom preservation* is a principle that says that any form that contains a bottom element is itself equal to bottom.

When an answer contains the variable *:bottom*, that answer has a circular structure, and the query that resulted in that answer should fail.

```
(define (preserve-bottom s)
  (cond ((occurs :bottom s s) ())
        (t s)))
```

Run* is the principal interface for submitting queries to amk:

```
(define (run* x g)
  (preserve-bottom
    (map (lambda (s)
           (walk* x (append s (reify (walk* x s)))))
         (g empty-s))))
```

X may be a logic variable or **()**.

When *x* is a variable, **run*** returns the value or values of that variable. When *x*=**()**, it returns either '(()) or **()**.

When a query fails, **run*** returns **()**.

Run* runs the goal *g* and then *walk**s each substitution of the answer.

It also reifies the fresh variables contained in each resulting substitution.

utilities

These are some predefined variables that were assumed to be defined in the previous sections.

```
(define vp (var 'p))
(define vq (var 'q))
```

The following code contains a collection of utility goals that belong to amk. They were discussed in detail earlier in this chapter.

```
(define (conso a d p) (== (cons a d) p))

(define (caro p a) (conso a (_) p))
```

```
(define (cdro p d) (conso (_) d p))

(define (pairo p) (conso (_) (_) p))

(define (eqo x y) (== x y))

(define (nullo a) (eqo a ()))

(define (memo x l)
  (let ((vt (var 't)))
    (any (caro l x)
         (all (cdro l vt)
              (lambda (s)
                ((memo x vt) s))))))

(define (appendo x y r)
  (any (all (== x ())
            (== y r))
       (let ((vh (var 'h))
             (vt (var 't))
             (va (var 'a)))
         (all (conso vh vt x)
              (conso vh va r)
              (lambda (s)
                ((appendo vt y va) s))))))

(define (memqo x l r)
  (let ((vt (var 't)))
    (any (all (caro l x)
              (== l r))
         (all (cdro l vt)
              (lambda (s)
                ((memqo x vt r) s))))))

(define (rmemqo x l r)
  (let ((vt (var 't)))
    (any (all (cdro l vt)
              (lambda (s)
                ((rmemqo x vt r) s)))
         (all (caro l x)
              (== l r)))))
```

part three zenlisp implementation

Zenlisp is an interpreter for the purely symbolic LISP language introduced and discussed in the earlier parts of this book. It is written in ANSI C (C89) and zenlisp. This part reproduces the complete source code of the zenlisp interpreter including lots of annotations. The first chapter will describe the about 3000 lines of C code and the second one the about 1000 lines of zenlisp code which together constitute the complete system.

12. c part

The complete C part of the source code is contained in the file `zl.c`.

```
/*
 * zenlisp -- an interpreter for symbolic LISP
 * By Nils M Holm <nmh@t3x.org>, 2007, 2008
 * Feel free to copy, share, and modify this program.
 * See the file LICENSE for details.
 */
```

Zenlisp is a tree-walking interpreter that implements shallow binding, a constant-space mark and sweep garbage collector, and bignum arithmetics (although these are considered a rather esoteric feature and are contained in the LISP part entirely.)

12.1 prelude and data declarations

```
#include <stdlib.h>
#ifdef __TURBOC__
 #include <io.h>
 #include <alloc.h>
#else
 #include <unistd.h>
 #ifndef __MINGW32__
  #ifndef __CYGWIN__
   #define setmode(fd, mode)
  #endif
 #endif
#endif
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <fcntl.h>

#define VERSION 2
#define RELEASE "2008-09-19"
```

`DEFAULT_NODES` is the default size of the **node pool** used by zenlisp. The node pool has a static size and cannot grow at run time. A different node size can be specified using a command line option. Larger node pools (up to some limit) typically mean faster operation and a bigger memory footprint.

Each *node* consists of "car" field, a "cdr" field, and a "tag" field. The first two have a size of an `int` each and the tag has a size of one `char`, so the total size of the node pool is computed as follows:

$\texttt{Size}_{\texttt{Pool}}$ `=` *Nodes* `* (2 * sizeof(int) + 1)`

The `MINIMUM_NODES` constant specifies the size of the smallest pool that can hold the LISP part of the system.

```
#define DEFAULT_NODES   131072
#define MINIMUM_NODES   12280
```

`DEFAULT_IMAGE` is the location of the LISP image file to load when the system starts up. A different image file can be specified on the command line.

```
#ifndef DEFAULT_IMAGE
 #define DEFAULT_IMAGE  "/u/share/zenlisp/zenlisp"
#endif
```

The `counter` structure is a portable mechanism for representing big numbers. It is used to gather data regarding allocation and reduction cycles at run time. Its `n` member stores ones, the `n1k` member thousands, etc.

```
struct counter {
        int     n, n1k, n1m, n1g;
};
```

The `Error_context` structure is used to store the context in which an error occurred, so the error may be reported at a later time. Each error message has the form

```
* file: line: function: message: expression
* additional argument
* Trace: function ...
```

The *file*, *expression*, *additional argument*, and *Trace: ...* parts are optional. The error context holds the individual parts of the message:

msg	error message
arg	additional argument
expr	expression that caused the error or `NO_EXPR`
file	input file or `NULL` for `stdin`
line	input line number
fun	function in which the error occurred or `NIL`
frame	current call frame, used to print trace

```
struct Error_context {
        char    *msg;
        char    *arg;
        int     expr;
        char    *file;
        int     line;
        int     fun;
        int     frame;
};
```

This is the maximum length of a symbol name, and the maximum length of a source path:

```
#define SYMBOL_LEN      256
#define MAX_PATH_LEN    256
```

The following flags are used to control the garbage collector. `ATOM_FLAG` is also used to distinguish atoms from pairs. Seriously, there are no types in zenlisp other than the atom (symbol) and the pair.

ATOM_FLAG	This node is an atom
MARK_FLAG	Used to tag live nodes during GC
SWAP_FLAG	Used to indicate that this node is not yet completely visited

When `SWAP_FLAG` is set, the "car" and "cdr" fields of the node will be swapped in order to visit the cdr child in garbage collections. Hence the name of this flag.

```
#define ATOM_FLAG       0x01
#define MARK_FLAG       0x02
#define SWAP_FLAG       0x04
```

Here are some magic values. `NIL` is the empty list. It spells out "not in list" and therefore its (internal) value is an invalid index of the node pool. `EOT` is the end of (input) text. It is used to denote the end of a file or TTY input. `DOT` and `R_PAREN` are delivered by the reader when a dot ("`.`") or a right parenthesis ("`)`") is found. `NO_EXPR` indicates that an error was not caused by any specific expression.

```
#define NIL     -1
#define EOT     -2
#define DOT     -3
#define R_PAREN -4
#define NO_EXPR -5
```

These are the states that the zenlisp evaluator may go through while reducing a program to its normal form. They will be explained in detail in the description and code of the `eval` function.

MATOM	evaluating an atom; this is the original state
MLIST	evaluating a list of function arguments
MBETA	evaluating the body of a function
MBIND	evaluating the bindings of **`let`**

MBINR	evaluating the bindings of **`letrec`**
MLETR	evaluating the term of **`let`** or **`letrec`**
MCOND	evaluating predicates of **`cond`**
MCONJ	evaluating expressions of **`and`** (but not the last one)
MDISJ	evaluating expressions of **`or`** (but not the last one)

```
enum Evaluator_states {
        MATOM = '0',
        MLIST,
        MBETA,
        MBIND,
        MBINR,
        MLETR,
        MCOND,
        MCONJ,
        MDISJ
};
```

The size of the node pool in nodes.

```
int     Pool_size;
```

The arrays `Car`, `Cdr`, and `Tag` form the node pool. `Car` and `Cdr` hold the car and cdr fields of a cons cell, `Tag` holds the atom flag and the garbage collector tags. The car field of a node **n** is referred to by `Car[n]`, the cdr field of the same node by `Cdr[n]`, and its tags are accessed using `Tag[n]`.

Note that no pointers are used inside of the node pool. Each car and cdr field (except for atoms) contains the offset of another node in the array forming the node pool. Figure 9 depicts the internal representation of a cons cell. The cell is located at offset 5, which means that `Car[5]` holds its car field and `Cdr[5]` holds its cdr field. Its car field contains the integer 17, so its car value is stored in the node located at offset 17. Analogously, the value of the cdr part of the cons cell is stored in the node located at offset 29.

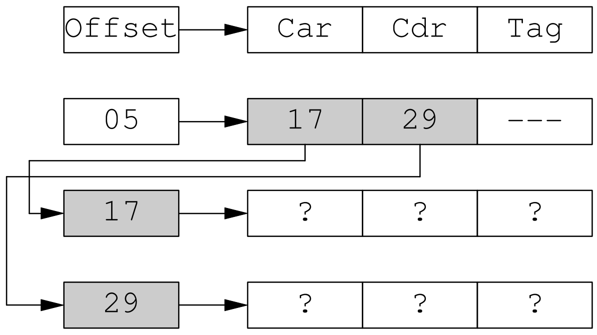

Fig. 9 – node pool structure

Because integer offsets are used for indirection instead of pointers, the entire node pool can be dumped to disk using basically three `write()` operations and restored at a later time using read(). All addressing is done relative to the pool arrays.

```
int     *Car,                   /* Car*Cdr*Tag = Node Pool */
        *Cdr;
char    *Tag;
```

This is a cdr-linked list of free nodes.

```
int     Freelist;
```

The following variables are protected during garbage collections. A value that is bound to any of them will not be recycled. Tmp_car and Tmp_cdr are used to protect child nodes at allocation time. The other two are used whenever they are needed.

```
int     Tmp_car, Tmp_cdr;       /* GC-safe */
int     Tmp, Tmp2;
```

Infile is the name of the input file currently being read. A value of NULL means that terminal input is read by the REPL (read-eval-print loop). Input is the input stream itself.

Rejected is a character that has been put back to the input stream. When Rejected=EOT, no character has been rejected. Of course, ungetc() could have been used, but this solution is more transparent and easier ported to other languages.[22]

Line is the current input line number relative to the beginning of the current input file. Output is the output stream to which all interpreter output (including error messages but excluding startup error messages) is sent.

```
char    *Infile;
FILE    *Input;
int     Rejected;
int     Line;
FILE    *Output;
```

The following string variables are used to buffer path names when **load**ing programs. Source_dir is the directory from which source files are loaded, Expanded_path is used to expand files names beginning with a tilde, and Current_path captures the current path in nested **load**s. See the load() function for details.

```
char    Source_dir[MAX_PATH_LEN];
char    Expanded_path[MAX_PATH_LEN];
char    Current_path[MAX_PATH_LEN];
```

Error_flag and Fatal_flag are set when a (fatal) error occurs.

```
int     Error_flag;
struct Error_context
        Error;
```

22 The ancestor of zenlisp was written in T3X and not in C, which may explain some decisions that appear un-C-ish.

```
int     Fatal_flag;
```

`Symbols` is the global symbol table. `Safe_symbols` is a copy of the symbol table that is used to store a sane copy in case things go awfully wrong during reduction.

Because zenlisp uses shallow binding (which stores values directly in variables), the symbol table is merely a list of symbols.

```
int     Symbols;
int     Safe_symbols;
```

These are the various stacks that the interpreter uses when reducing expressions to their normal forms:

Stack	general-purpose stack
Stack_bottom	bottom of `Stack`, because `eval()` is reentrant
Mode_stack	interpreter states
Arg_stack	function arguments
Bind_stack	bindings of **`let`** and **`letrec`**
Env_stack	lexical environments of closures (performance hack)

```
int     Stack, Stack_bottom;
int     Mode_stack;
int     Arg_stack;
int     Bind_stack;
int     Env_stack;
```

These variables are used in error reporting and debugging:

```
int     Frame;
int     Function_name;
int     Traced_fn;
```

`Root` contains all variables whose values are to be protected during garbage collections. The collector will never recycle a value that is bound to any of these variables.

```
int     *Root[] = { &Symbols, &Stack, &Mode_stack, &Arg_stack, &Bind_stack,
                    &Env_stack, &Tmp_car, &Tmp_cdr, &Tmp, &Tmp2,
                    &Safe_symbols, NULL };
```

These variables are used to capture lexical environments. `Lexical_env` holds the lexical environment to build. `Bound_vars` contains the list of variables bound in a given context.

```
int     Lexical_env;
int     Bound_vars;
```

The next variables are used to keep track of the nesting levels of parentheses in the input as well as applications of **`load`** and `eval()`. The `eval()` function recurses internally during program reduction, but some functions (like **`define`**) are limited to the top level of recursion.

```
int     Paren_level;
int     Load_level;
int     Eval_level;
```

When `Quotedprint` is set to one, printed expressions are assumed to be quoted already, so no leading quote (apostrophy) will print. `Max_atoms_used` records the peak of the node usage during reduction. It is cleared by the **gc** function. `Max_trace` holds the maximal number of function names to print in the call trace in case of an error.

```
int     Quotedprint;
int     Max_atoms_used;
int     Max_trace;
```

When `Stat_flag` is set, reduction steps, nodes allocated, and garbage collections are counted. It is used internally by the **stats** pseudo function. `Closure_form` determines how much of a closure will print. 0 means only the arguments, 1 includes the body, 2 includes the lexical environment. `Verify_arrows` turns arrow verification on and off. `Verbose_GC` controls the output of GC statistics.

```
int     Stat_flag;
int     Closure_form;
int     Verify_arrows;
int     Verbose_GC;
```

These are counters for the **stats** pseudo function.

```
struct counter  Reductions,
                Allocations,
                Collections;
```

The following variables hold the offsets of frequently used symbols, so the symbols do not have to be looked up in the symbol table each time they are referred to. Some of the symbols are internal, some are zenlisp keywords.

```
int     S_bottom, S_closure, S_false, S_lambda, S_primitive,
        S_quote, S_special, S_special_cbv, S_true, S_void,
        S_last;
```

These are the opcodes of zenlisp's primitive functions. They are offsets to the `Primitives` array, which holds pointers to the functions implementing the associated operations.

```
enum {  P_ATOM, P_BOTTOM, P_CAR, P_CDR, P_CONS, P_DEFINED, P_EQ,
        P_EXPLODE, P_GC, P_IMPLODE, P_QUIT, P_RECURSIVE_BIND,
        P_SYMBOLS, P_VERIFY_ARROWS, N_PRIMITIVES };

int     (*Primitives[N_PRIMITIVES])(int);
```

Pseudo function applications (special forms) are handled in the same way as primitive functions.

```
enum {  SF_AND, SF_APPLY, SF_CLOSURE_FORM, SF_COND, SF_DEFINE,
        SF_DUMP_IMAGE, SF_EVAL, SF_LAMBDA, SF_LET, SF_LETREC,
        SF_LOAD, SF_OR, SF_QUOTE, SF_STATS, SF_TRACE,
        N_SPECIALS };

int     (*Specials[N_SPECIALS])(int, int *, int *, int *);
```

Stop lint from complaining about unused variables.

```
#ifdef LINT
  #define USE(arg)      (arg = NIL)
#else
  #define USE(arg)
#endif
```

Every single function in the program has a prototype. Feel free to skip ahead.

```
int     _rdch(void);
int     add_primitive(char *name, int opcode);
int     add_special(char *name, int opcode, int cbv);
int     add_symbol(char *s, int v);
int     alloc3(int pcar, int pcdr, int ptag);
int     aunsave(int k);
int     bad_argument_list(int n);
void    bind_args(int n, int name);
int     bunsave(int k);
void    catch_int(int sig);
void    clear_stats(void);
void    collect_free_vars(int n);
int     cond_get_pred(void);
int     cond_eval_clause(int n);
int     cond_setup(int n);
int     copy_bindings(void);
void    count(struct counter *c, int k);
char    *counter_to_string(struct counter *c, char *buf);
int     define_function(int n);
int     dump_image(char *p);
int     equals(int n, int m);
void    eliminate_tail_calls(void);
int     error(char *m, int n);
int     eval(int n);
char    *expand_path(char *s, char *buf);
int     explode_string(char *sym);
void    fatal(char *m);
int     find_symbol(char *s);
void    fix_all_closures(int b);
void    fix_cached_closures(void);
void    fix_closures_of(int n, int bindings);
int     flat_copy(int n, int *lastp);
int     gc(void);
int     get_opt_val(int argc, char **argv, int *pi, int *pj, int *pk);
void    get_options(int argc, char **argv);
void    get_source_dir(char *path, char *pfx);
char    *symbol_to_string(int n, char *b, int k);
```

```
void    help(void);
void    init(void);
void    init1(void);
void    init2(void);
int     is_alist(int n);
int     is_bound(int n);
int     is_list_of_symbols(int m);
void    let_bind(int env);
int     let_eval_arg(void);
int     let_finish(int rec);
int     let_next_binding(int n);
int     let_setup(int n);
int     load(char *p);
int     make_closure(int n);
void    mark(int n);
int     make_lexical_env(int term, int locals);
char    *make_zen_path(char *s);
int     munsave(void);
void    nl(void);
void    print(int n);
int     reverse_in_situ(int n);
void    pr(char *s);
int     primitive(int *np);
void    print_call_trace(int n);
int     print_closure(int n, int dot);
int     print_condensed_list(int n, int dot);
int     print_primitive(int n, int dot);
int     print_quoted_form(int n, int dot);
void    print_trace(int n);
void    print_license(void);
void    pr_num(int n);
int     quote(int n);
int     read_condensed(void);
void    read_eval_loop(void);
int     read_list(void);
int     read_symbol(int c);
void    repl(void);
void    reset_counter(struct counter *c);
void    reset_state(void);
void    restore_bindings(int values);
int     setup_and_or(int n);
int     special(int *np, int *pcf, int *pmode, int *pcbn);
int     string_to_symbol(char *s);
char    *symbol_to_string(int n, char *b, int k);
void    unbind_args(void);
int     unreadable(void);
int     unsave(int k);
void    usage(void);
void    verify(void);
int     wrong_arg_count(int n);
int     z_and(int n, int *pcf, int *pmode, int *pcbn);
int     z_apply(int n, int *pcf, int *pmode, int *pcbn);
int     z_atom(int n);
int     z_bottom(int n);
int     z_car(int n);
int     z_cdr(int n);
int     z_closure_form(int n, int *pcf, int *pmode, int *pcbn);
int     z_cond(int n, int *pcf, int *pmode, int *pcbn);
int     z_cons(int n);
int     z_define(int n, int *pcf, int *pmode, int *pcbn);
int     z_defined(int n);
int     z_dump_image(int n, int *pcf, int *pmode, int *pcbn);
int     z_eq(int n);
int     z_eval(int n, int *pcf, int *pmode, int *pcbn);
int     z_explode(int n);
int     z_gc(int n);
int     z_implode(int n);
int     z_lambda(int n, int *pcf, int *pmode, int *pcbn);
int     z_let(int n, int *pcf, int *pmode, int *pcbn);
int     z_letrec(int n, int *pcf, int *pmode, int *pcbn);
int     z_load(int n, int *pcf, int *pmode, int *pcbn);
int     z_or(int n, int *pcf, int *pmode, int *pcbn);
int     z_quit(int n);
int     z_quote(int n, int *pcf, int *pmode, int *pcbn);
int     z_recursive_bind(int n);
int     z_stats(int n, int *pcf, int *pmode, int *pcbn);
int     z_symbols(int n);
int     z_trace(int n, int *pcf, int *pmode, int *pcbn);
int     z_verify_arrows(int n);
int     zen_eval(int n);
void    zen_fini(void);
int     zen_init(int nodes, int trackGc);
char    **zen_license(void);
int     zen_load_image(char *p);
void    zen_print(int n);
void    zen_print_error(void);
int     zen_read(void);
void    zen_stop(void);
int     zread(void);
```

12.2 miscellaneous functions

These are convenience macros for accessing members of nested lists.

```
#define caar(x) (Car[Car[x]])
#define cadr(x) (Car[Cdr[x]])
#define cdar(x) (Cdr[Car[x]])
#define cddr(x) (Cdr[Cdr[x]])
#define caaar(x) (Car[Car[Car[x]]])
#define caadr(x) (Car[Car[Cdr[x]]])
#define cadar(x) (Car[Cdr[Car[x]]])
#define caddr(x) (Car[Cdr[Cdr[x]]])
```

```
#define cdaar(x) (Cdr[Car[Car[x]]])
#define cddar(x) (Cdr[Cdr[Car[x]]])
#define cdddr(x) (Cdr[Cdr[Cdr[x]]])
#define caddar(x) (Car[Cdr[Cdr[Car[x]]]])
#define cadddr(x) (Car[Cdr[Cdr[Cdr[x]]]])
```

All interpreter output (except for startup error messages) goes through this interface.

```
void nl(void) {
        putc('\n', Output);
        if (Output == stdout) fflush(Output);
}

void pr(char *s) {
        fputs(s, Output);
}

void pr_num(int n) {
        fprintf(Output, "%d", n);
}
```

12.3 error reporting

`Print_call_trace` prints a call trace if a non-empty call frame is passed to it.

```
void print_call_trace(int frame) {
        int     s, n;

        s = frame;
        n = Max_trace;
        while (s != NIL) {
                if (n == 0 || Cdr[s] == NIL || cadr(s) == NIL) break;
                if (n == Max_trace) pr("* Trace:");
                n = n-1;
                pr(" ");
                Quotedprint = 1;
                print(cadr(s));
                s = Car[s];
        }
        if (n != Max_trace) nl();
}
```

Register an error context for later reporting and set the error flag. In case of multiple errors, register only the first one.

```
int error(char *m, int n) {
        if (Error_flag) return NIL;
        Error.msg = m;
        Error.expr = n;
        Error.file = Infile;
        Error.line = Line;
        Error.fun = Function_name;
        Error.frame = Frame;
        Error_flag = 1;
        return NIL;
}
```

Print the error message currently stored in the error context and clear the error flag.

```
void zen_print_error(void) {
        pr("* ");
        if (Error.file) {
                pr(Error.file);
                pr(": ");
        }
        pr_num(Error.line);
        pr(": ");
        if (Error.fun != NIL) {
                Quotedprint = 1;
                print(Error.fun);
        }
        else {
                pr("REPL");
        }
        pr(": ");
        pr(Error.msg);
        if (Error.expr != NO_EXPR) {
                if (Error.msg[0]) pr(": ");
                Quotedprint = 1;
                print(Error.expr);
        }
        nl();
        if (Error.arg) {
                pr("* ");
                pr(Error.arg); nl();
                Error.arg = NULL;
        }
        if (!Fatal_flag && Error.frame != NIL)
                print_call_trace(Error.frame);
        Error_flag = 0;
}
```

Report a fatal error and exit.

```
void fatal(char *m) {
        Error_flag = 0;
        Fatal_flag = 1;
        error(m, NO_EXPR);
        zen_print_error();
        pr("* Fatal error, aborting");
        nl();
        exit(1);
}
```

12.4 counting functions

```
void reset_counter(struct counter *c) {
        c->n = 0;
        c->n1k = 0;
        c->n1m = 0;
        c->n1g = 0;
}
```

Increment the counter **c** by **k**. Assert 0<=**k**<=1000.

```
void count(struct counter *c, int k) {
        char    *msg = "statistics counter overflow";

        c->n = c->n+k;
        if (c->n >= 1000) {
                c->n = c->n - 1000;
                c->n1k = c->n1k + 1;
                if (c->n1k >= 1000) {
                        c->n1k = 0;
                        c->n1m = c->n1m+1;
                        if (c->n1m >= 1000) {
                                c->n1m = 0;
                                c->n1g = c->n1g+1;
                                if (c->n1g >= 1000) {
                                        error(msg, NO_EXPR);
                                }
                        }
                }
        }
}
```

Convert a counter structure to a string representation of the value stored in it. Commas will be inserted to mark thousands. A sufficiently large string buffer must be supplied by the caller. The greatest value that can be stored in a counter structure is 999,999,999,999.

```
char *counter_to_string(struct counter *c, char *buf) {
        int     i;

        i = 0;
        if (c->n1g) {
                sprintf(&buf[i], "%d,", c->n1g);
                i = strlen(buf);
        }
        if (c->n1m || c->n1g) {
                if (c->n1g)
                        sprintf(&buf[i], "%03d,", c->n1m);
                else
                        sprintf(&buf[i], "%d,", c->n1m);
                i = strlen(buf);
        }
        if (c->n1k || c->n1m || c->n1g) {
                if (c->n1g || c->n1m)
                        sprintf(&buf[i], "%03d,", c->n1k);
                else
                        sprintf(&buf[i], "%d,", c->n1k);
                i = strlen(buf);
        }
        if (c->n1g || c->n1m || c->n1k)
                sprintf(&buf[i], "%03d", c->n);
        else
                sprintf(&buf[i], "%d", c->n);
        return buf;
}
```

12.5 memory management

The `mark()` function implements a finite state machine (FSM) that traverses a tree rooted at the node `n`. The function marks all nodes that it encounters during traversal as ‘‘live’’ nodes, i.e. nodes that may not be recycled. The FSM uses three states (1,2,3) that are formed using the collector flags `MARK_FLAG` (M) and `SWAP_FLAG` (S). `MARK_FLAG` is a state flag and the ‘‘mark’’ flag — which is used to tag live nodes — at the same time. The following figures illustrate the states of the root node during the traversal of a tree of three nodes. Marked nodes are rendered with a grey background.

State 1: Node **N** is unvisited. The parent points to NIL, both flags are cleared.

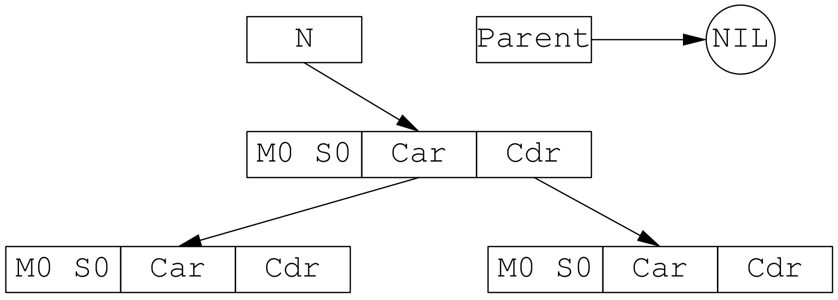

Fig. 10 – garbage collection, state 1

State 2: **N** now points to the car child of the root node, the parent pointer points to the root node, and the parent of the parent is stored in the car part of the root node. Both flags are set. The node is now marked.

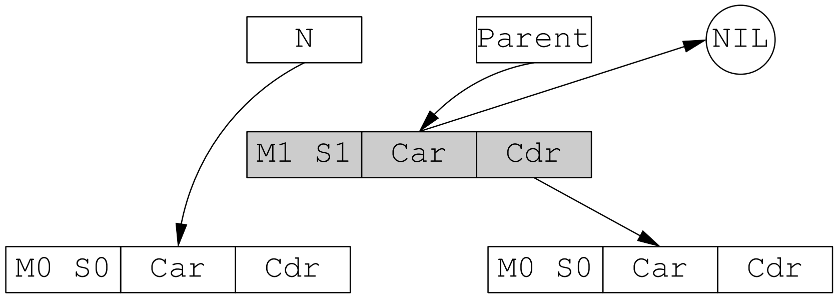

Fig. 11 – garbage collection, state 2

State 3: When the car child is completed, the car pointer of the root is restored, the grandparent moves to the cdr part of the root node, and **N** moves to the cdr child. The **S** flag is cleared, and the root node is now completely traversed.

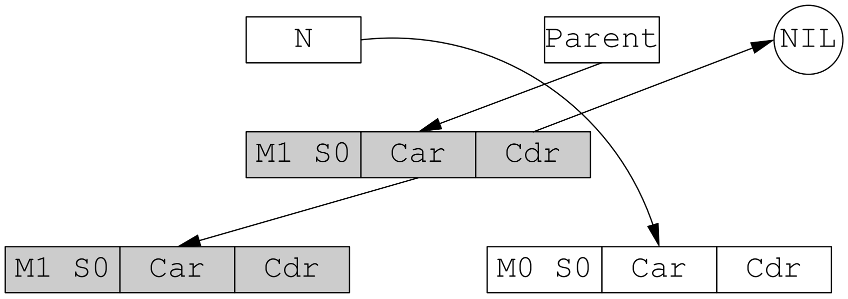

Fig. 12 – garbage collection, state 3

State 3: When the FSM returns from the cdr child, it finds the root node in state 3. To return to the root, it restores the cdr pointer of the root node and the parent. **N** moves up to the root node. Because **N** is marked and parent is NIL, the traversal is complete.

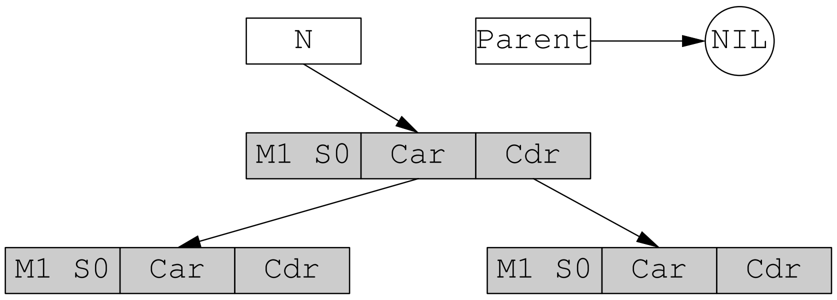

Fig. 13 – garbage collection, finished

When the FSM hits an already marked node during traversal, it returns to the parent node immediately. Because nodes get marked *before* their descendants are traversed, the FSM can traverse cyclic structures without entering an infinite loop.

When the mark phase finds an object with the `ATOM_FLAG` set, it traverses only its cdr field and leaves the car field alone (because it does not hold a valid node offset).

```
/*
 * Mark nodes which can be accessed through N.
 * Using the Deutsch/Schorr/Waite (aka pointer reversal) algorithm.
 * State 1: M==0 S==0 unvisited, process CAR
 * State 2: M==1 S==1 CAR visited, process CDR
 * State 3: M==1 S==0 completely visited, return to parent
 */
void mark(int n) {
        int     p, parent;

        parent = NIL;
        while (1) {
                if (n == NIL || Tag[n] & MARK_FLAG) {
                        if (parent == NIL) break;
                        if (Tag[parent] & SWAP_FLAG) {      /* State 2 */
                                /* Swap CAR and CDR pointers and */
                                /* proceed with CDR. Go to State 3. */
                                p = Cdr[parent];
                                Cdr[parent] = Car[parent];
                                Car[parent] = n;
                                Tag[parent] &= ~SWAP_FLAG;      /* S=0 */
                                Tag[parent] |=  MARK_FLAG;      /* M=1 */
                                n = p;
                        }
                        else {                              /* State 3: */
                                /* Return to the parent and restore */
                                /* parent of parent */
                                p = parent;
                                parent = Cdr[p];
                                Cdr[p] = n;
                                n = p;
                        }
                }
                else {                                      /* State 1: */
                        if (Tag[n] & ATOM_FLAG) {
                                /* If this node is an atom, go directly */
                                /* to State 3. */
                                p = Cdr[n];
                                Cdr[n] = parent;
                                /*Tag[n] &= ~SWAP_FLAG;*/       /* S=0 */
                                parent = n;
                                n = p;
                                Tag[parent] |= MARK_FLAG;       /* M=1 */
                        }
                        else {
                                /* Go to state 2: */
                                p = Car[n];
                                Car[n] = parent;
                                Tag[n] |= MARK_FLAG;            /* M=1 */
                                parent = n;
                                n = p;
                                Tag[parent] |= SWAP_FLAG;       /* S=1 */
                        }
                }
        }
}
```

Mark and Sweep garbage collector: first mark all `Root[]` nodes and the nodes of the error context (if necessary), then delete and rebuild the free list. Mark flags are cleared in the loop that builds the new free list.

```
int gc(void) {
        int     i, k;

        k = 0;
        for (i=0; Root[i]; i++) mark(Root[i][0]);
        if (Error_flag) {
                mark(Error.expr);
                mark(Error.fun);
                mark(Error.frame);
        }
        Freelist = NIL;
        for (i=0; i<Pool_size; i++) {
                if (!(Tag[i] & MARK_FLAG)) {
                        Cdr[i] = Freelist;
                        Freelist = i;
                        k = k+1;
                }
                else {
                        Tag[i] &= ~MARK_FLAG;
                }
        }
        if (Max_atoms_used < Pool_size-k) Max_atoms_used = Pool_size-k;
        if (Verbose_GC) {
                pr_num(k);
                pr(" nodes reclaimed");
                nl();
        }
        if (Stat_flag) count(&Collections, 1);
        return k;
}
```

The `alloc3()` function is the principal node allocator of zenlisp. It removes the first node of the free list and initializes it with the given car, cdr, and tag values. When the free list is empty, a garbage collection (GC) is triggered. Note that `alloc3()` protects the values passed to it from the GC, so no special care has to be taken by the caller. For example, the form `(x y z)` can be created using the code fragment:

```
                        n = alloc(z, NIL);
                        n = alloc(y, n);
                        n = alloc(x, n);
```

```
int alloc3(int pcar, int pcdr, int ptag) {
        int     n;

        if (Stat_flag) count(&Allocations, 1);
        if (Freelist == NIL) {
                Tmp_cdr = pcdr;
                if (!ptag) Tmp_car = pcar;
                gc();
                Tmp_car = Tmp_cdr = NIL;
                if (Freelist == NIL) fatal("alloc3(): out of nodes");
        }
        n = Freelist;
        Freelist = Cdr[Freelist];
        Car[n] = pcar;
        Cdr[n] = pcdr;
        Tag[n] = ptag;
        return n;
}
```

`Alloc()` is a short cut for allocating cons cells.

```
#define alloc(pcar, pcdr) \
        alloc3(pcar, pcdr, 0)
```

`Save()` saves a node on the stack, `unsave()` removes a given number of values and returns the deepest one removed by it.

```
#define save(n) \
        (Stack = alloc(n, Stack))

int unsave(int k) {
        int     n;

        USE(n);
        while (k) {
                if (Stack == NIL) fatal("unsave(): stack underflow");
                n = Car[Stack];
                Stack = Cdr[Stack];
                k = k-1;
        }
        return n;
}
```

`Msave()` and `munsave()` work like `save()` and `unsave()` above, but use the mode stack rather than the general purpose stack. Because `Mode_stack` holds integer values instead of nodes, the values are packaged in the car fields of atom nodes.

```
#define msave(v) \
        (Car[Mode_stack] = alloc3(v, Car[Mode_stack], ATOM_FLAG))

int munsave(void) {
        int     v;

        if (Car[Mode_stack] == NIL) fatal("munsave(): m-stack underflow");
        v = caar(Mode_stack);
        Car[Mode_stack] = cdar(Mode_stack);
        return v;
}
```

The following functions repeat the save/unsave procedure for the argument stack and binding stack, respectively.

```
#define asave(n) \
        (Arg_stack = alloc(n, Arg_stack))

int aunsave(int k) {
        int     n;

        USE(n);
        while (k) {
                if (Arg_stack == NIL) fatal("aunsave(): a-stack underflow");
                n = Car[Arg_stack];
                Arg_stack = Cdr[Arg_stack];
                k = k-1;
        }
        return n;
}

#define bsave(n) \
        (Bind_stack = alloc(n, Bind_stack))

int bunsave(int k) {
        int     n;

        USE(n);
        while (k) {
                if (Bind_stack == NIL) fatal("bunsave(): b-stack underflow");
                n = Car[Bind_stack];
                Bind_stack = Cdr[Bind_stack];
                k = k-1;
        }
        return n;
}
```

12.6 symbol tables

Find a symbol with the name **s** in the global symbol table. Return the symbol or `NIL` if no such symbol exists.

```
int find_symbol(char *s) {
        int     p, n, i;

        p = Symbols;
        while (p != NIL) {
                n = caar(p);
                i = 0;
                while (n != NIL && s[i]) {
                        if (s[i] != (Car[n] & 255)) break;
                        n = Cdr[n];
                        i = i+1;
                }
                if (n == NIL && !s[i]) return Car[p];
                p = Cdr[p];
        }
        return NIL;
}
```

Check whether a node is an atom in the sense of **`atom`**. *Note:* **`atom`** also classifies primitive functions, special form handlers and the **`{void}`** value as atoms. This is not covered by the `atomic()` function.

```
#define atomic(n) \
        ((n) == NIL || (Car[n] != NIL && (Tag[Car[n]] & ATOM_FLAG)))
```

The `symbolic()` function checks whether the given node represents a symbol. This is probably the right place to clarify what the difference between *atoms*, *symbols*, and *atomic nodes* is.

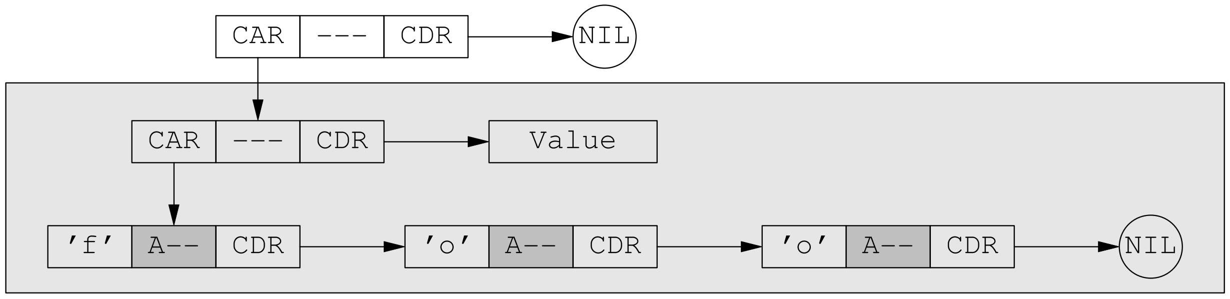

Fig. 14 – symbols, atoms, and atomic nodes

Figure 14 shows a list containing the symbol *foo* in so-called *box notation.* Each box containing three smaller boxes represents a node and each of the smaller boxes represents one field of that node. The first smaller box contains the car field, the second one the tag field, and the last one the cdr field. Three minus signs in a tag field indicate that no tags are set.

In Figure 14, the node outside of the large grey box is the "spine" of the form `(foo)`. Its cdr part points to **()** and its car part points to the symbol *foo.* Note that the symbol consists of (at least) four nodes: one that binds the symbol name to the value and three nodes that hold the characters of the symbol name.

A node with its `ATOM_FLAG` set is called an "atomic node". It is used to hold some value, like a character of a symbol name, in its car field. A "symbol" is a node whose car part points to a chain of atomic nodes and whose cdr part points to a node tree that represents the value of that symbol. An "atom", finally, is either a symbol or a primitive function, or **()**.

The `symbolic()` function checks whether a node is a symbol.

```
#define symbolic(n) \
        ((n) != NIL && Car[n] != NIL && (Tag[Car[n]] & ATOM_FLAG))
```

Create a symbol with the given name and return it.

```
int string_to_symbol(char *s) {
        int     i, n, m, a;

        i = 0;
        if (s[i] == 0) return NIL;
        a = n = NIL;
        while (s[i]) {
                m = alloc3(s[i], NIL, ATOM_FLAG);
                if (n == NIL) {
                        n = m;
                        save(n);
                }
                else {
                        Cdr[a] = m;
                }
                a = m;
                i = i+1;
        }
        unsave(1);
        return n;
}
```

Create a string containing the name of the given symbol. When the symbol has a length of more then SYMBOL_LEN characters, an error is reported.

```
char *symbol_to_string(int n, char *b, int k) {
        int     i;

        n = Car[n];
        for (i=0; i<k-1; i++) {
                if (n == NIL) break;
                b[i] = Car[n];
                n = Cdr[n];
        }
        if (n != NIL) {
                error("symbol_to_string(): string too long", NO_EXPR);
                return NULL;
        }
        b[i] = 0;
        return b;
}
```

Add a symbol to the global symbol table. If a symbol with the given name already exists, return it without creating a new one.

```
int add_symbol(char *s, int v) {
        int     n, m;

        n = find_symbol(s);
        if (n != NIL) return n;
        n = string_to_symbol(s);
        m = alloc(n, v? v: n);
        Symbols = alloc(m, Symbols);
        return m;
}
```

Add a primitive function (add_primitive()) or special form handler (add_special()) to the current symbol table. Figure 15 outlines the internal structure of a primitive function.

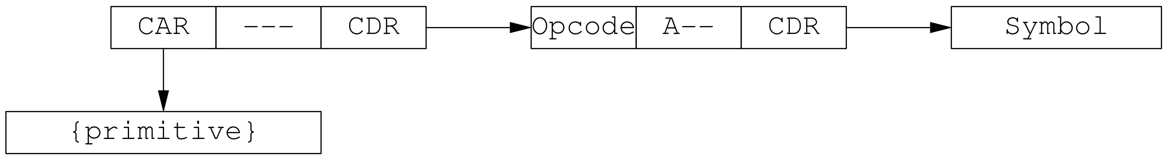

Fig. 15 – primitive function structure

The special symbol **`{primitive}`** marks the structure as a primitive function. The atom contains the opcode of the primitive and a link back to the name to which the primitive is bound. This allows the printer to output `{internal car}` rather than just `{internal}` when evaluating **`car`**.

Special form (SF) handlers have the same structure as primitive function handlers, but they use the **`{special}`** or **`{special_cbv}`** symbols instead of **`{primitive}`**. SF handlers using the **`{special_cbv}`** symbol are called by value. Other SF handlers are called by name.

```
int add_primitive(char *name, int opcode) {
        int     y;

        y = add_symbol(name, 0);
        Cdr[y] = alloc(S_primitive, NIL);
        cddr(y) = alloc3(opcode, NIL, ATOM_FLAG);
        cdddr(y) = y;
        return y;
}

int add_special(char *name, int opcode, int cbv) {
        int     y;

        y = add_symbol(name, 0);
        Cdr[y] = alloc(cbv? S_special_cbv: S_special, NIL);
        cddr(y) = alloc3(opcode, NIL, ATOM_FLAG);
        cdddr(y) = y;
        return y;
}
```

12.7 reader

All interpreter input goes through this interface.

```
int _rdch(void) {
        int     c;

        if (Rejected != EOT) {
                c = Rejected;
                Rejected = EOT;
                return c;
        }
        c = getc(Input);
        if (feof(Input)) return EOT;
        if (c == '\n') Line = Line+1;
        return c;
}

#define rdch() \
        tolower(_rdch())
```

The `read_list()` function reads proper lists and improper lists (including, of course, dotted pairs). It calls `zread()` to read each member of the list. `Zread()` may call `read_list()` in turn to read sublists. When `read_list()` is invoked, the initial opening parenthesis already has been removed from the input.

`Read_read()` checks for properly formed improper lists (sic!) and reports all kinds of errors, like missing closing parentheses and dots in unexpected positions. It returns the list read in case of success and otherwise **`()`**.

```
int read_list(void) {
        int     n,
                lst,
                app,
                count;
        char    *badpair;

        badpair = "bad pair";
        Paren_level = Paren_level+1;
        lst = alloc(NIL, NIL);  /* Root node */
        save(lst);
        app = NIL;
        count = 0;
        while (1) {
                if (Error_flag) {
                        unsave(1);
                        return NIL;
                }
                n = zread();
                if (n == EOT)  {
                        if (Load_level) return EOT;
                        error("missing ')'", NO_EXPR);
                }
                if (n == DOT) {
                        if (count < 1) {
                                error(badpair, NO_EXPR);
                                continue;
                        }
                        n = zread();
                        Cdr[app] = n;
                        if (n == R_PAREN || zread() != R_PAREN) {
                                error(badpair, NO_EXPR);
                                continue;
                        }
                        unsave(1);
                        Paren_level = Paren_level-1;
                        return lst;
                }
                if (n == R_PAREN) break;
                if (app == NIL)
                        app = lst;
                else
                        app = Cdr[app];
                Car[app] = n;
                Cdr[app] = alloc(NIL, NIL);
                count = count+1;
        }
        Paren_level = Paren_level-1;
        if (app != NIL) Cdr[app] = NIL;
        unsave(1);
        return count? lst: NIL;
}
```

This function checks whether a given character is a *delimiter*. Delimiters separate individual tokens in the input stream.

```
#define is_delimiter(c) \
                ((c) == ' ' || \
                 (c) == '\t' || \
                 (c) == '\n' || \
                 (c) == '\r' || \
                 (c) == '(' || \
                 (c) == ')' || \
                 (c) == ';' || \
                 (c) == '.' || \
                 (c) == '#' || \
                 (c) == '{' || \
                 (c) == '\'')
```

Read a condensed list (excluding the introducing "#" character) and return it.

```
int read_condensed(void) {
        int     n, c, a;
        char    s[2];

        n = alloc(NIL, NIL);
        save(n);
        a = NIL;
        s[1] = 0;
        c = rdch();
        while (!is_delimiter(c)) {
                if (a == NIL) {
                        a = n;
                }
                else {
                        Cdr[a] = alloc(NIL, NIL);
                        a = Cdr[a];
                }
                s[0] = c;
                Car[a] = add_symbol(s, S_void);
                c = rdch();
        }
        unsave(1);
        Rejected = c;
        return n;
}
```

Convert a string to a list of single-character symbols (a condensed list). Return the resulting list. This function is used internally to convert numeric values to a form that can be printed by the zenlisp printer, e.g. "12345" —> `#12345`.

```
int explode_string(char *sym) {
        int     n, a, i;
        char    s[2];

        n = alloc(NIL, NIL);
        save(n);
        a = NIL;
        s[1] = 0;
        i = 0;
        while (sym[i]) {
                if (a == NIL) {
                        a = n;
                }
                else {
                        Cdr[a] = alloc(NIL, NIL);
                        a = Cdr[a];
                }
                s[0] = sym[i];
                Car[a] = add_symbol(s, S_void);
                i += 1;
        }
        unsave(1);
        return n;
}
```

Quote the node **n**.

```
int quote(int n) {
        int     q;

        q = alloc(n, NIL);
        return alloc(S_quote, q);
}
```

Read a symbol and return it. When the symbol is not yet the symbol table, add it.

```
int read_symbol(int c) {
        char    s[SYMBOL_LEN];
        int     i;

        i = 0;
        while (!is_delimiter(c)) {
                if (i >= SYMBOL_LEN-2) {
                        error("symbol too long", NO_EXPR);
                        i = i-1;
                }
                s[i] = c;
                i = i+1;
                c = rdch();
        }
        s[i] = 0;
        Rejected = c;
        return add_symbol(s, S_void);
}
```

Check whether two forms are equal. This function is equal to the **`equal`** function of zenlisp. Because **`equal`** is written in zenlisp, it is not yet available at this point.

```
int equals(int n, int m) {
        if (n == m) return 1;
        if (n == NIL || m == NIL) return 0;
        if (Tag[n] & ATOM_FLAG || Tag[m] & ATOM_FLAG) return 0;
        return equals(Car[n], Car[m])
            && equals(Cdr[n], Cdr[m]);
}
```

The `verify()` function reads a form that follows a **`=>`** operator and compares it to the normal form of the most recently reduced expression. When the forms differ, it reports an error.

```
void verify(void) {
        int     expected;

        expected = zread();
        if (!atomic(expected) && Car[expected] == S_quote)
                expected = cadr(expected);
        if (!equals(expected, Cdr[S_last]))
                error("Verification failed; expected", expected);
}
```

Read an unreadable object (sic!) and report it.

```
int unreadable(void) {
        #define L 256
        int             c, i;
        static char     b[L];

        i = 0;
        b[0] = '{';
        c = '{';
        while (c != '}' && c != EOT && i < L-2) {
                b[i++] = c;
                c = rdch();
        }
        b[i] = '}';
        b[i+1] = 0;
        Error.arg = b;
        return error("unreadable object", NO_EXPR);
}
```

`Zread()` is the zenlisp reader interface. It reads a form from the input stream and returns a tree of nodes that is the internal representation of that form. It also skips over comments and evaluates **=>** operators. When the end of the input stream has been reached, calling `zread()` yields `EOT`.

```
int zread(void) {
        int     c;

        c = rdch();
        while (1) {
                while (c == ' ' || c == '\t' || c == '\n' || c == '\r') {
                        if (Error_flag) return NIL;
                        c = rdch();
                }
                if (c == '=' && Paren_level == 0) {
                        c = rdch();
                        if (c != '>') {
                                Rejected = c;
                                c = '=';
                                break;
                        }
                        if (Verify_arrows) verify();
                }
                else if (c != ';') {
                        break;
                }
                while (c != '\n') c = rdch();
        }
        if (c == EOT) return EOT;
        if (c == '(') {
                return read_list();
        }
        else if (c == '\'') {
                return quote(zread());
        }
        else if (c == '#') {
                return read_condensed();
        }
        else if (c == ')') {
                if (!Paren_level) return error("unexpected ')'", NO_EXPR);
                return R_PAREN;
        }
        else if (c == '.') {
                if (!Paren_level) return error("unexpected '.'", NO_EXPR);
                return DOT;
        }
        else if (c == '{') {
                return unreadable();
        }
        else {
                return read_symbol(c);
        }
}
```

12.8 primitive operation handlers

A *primitive function* of zenlisp is a LISP function that is implemented in C. Its implementation is called a "*primitive operation handler*".

This section discusses the primitive functions of zenlisp. Each primitive operation handler receives an expression and returns its value. The arguments of primitives already are in their normal forms at this point.

These functions are short cuts for reporting common errors.

```
int wrong_arg_count(int n) {
        return error("wrong argument count", n);
}

int bad_argument_list(int n) {
        return error("bad argument list", n);
}
```

Zenlisp prototypes like the **cons** prototype below have the following general form:

(function argument$_1$ ...) ⟶ result$_1$ | result$_2$

Function is the described function, each **argument** specifies the type of one argument, **result** is the type of the resulting normal form, and **...** indicates that zero or more instances of the preceding datum may occur. The arrow denotes a mapping from the argument types to the type of the normal form. The vertical bar ("|") represents a logical "or". It may occur in argument lists, too.

Note that prototypes are not part of the zenlisp language. They are only used to describe functions in a formal way.

(cons form form) ⟶ pair

```
int z_cons(int n) {
        int     m, m2;
        m = Cdr[n];
        if (m == NIL || Cdr[m] == NIL || cddr(m) != NIL)
                return wrong_arg_count(n);
        m2 = cadr(m);
        m = alloc(Car[m], m2);
        return m;
}
```

(car pair) ⟶ form

```
int z_car(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        m = Car[m];
        if (    atomic(m) ||
                Car[m] == S_primitive ||
                Car[m] == S_special ||
                Car[m] == S_special_cbv
        )
                return error("car: cannot split atoms", m);
        return Car[m];
}
```

(cdr pair) ⟶ form

```
int z_cdr(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        m = Car[m];
        if (    atomic(m) ||
                Car[m] == S_primitive ||
                Car[m] == S_special ||
                Car[m] == S_special_cbv
        )
                return error("cdr: cannot split atoms", m);
        return Cdr[m];
}
```

(eq form$_1$ form$_2$) ⟶ :t | :f

```
int z_eq(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] == NIL || cddr(m) != NIL)
                return wrong_arg_count(n);
        return Car[m] == cadr(m)? S_true: S_false;
}
```

(atom form) ⟶ :t | :f

Note that **atom** also returns **:t** for primitive functions, special form handlers and **{void}**.

```
int z_atom(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        if atomic(Car[m]) return S_true;
        m = caar(m);
        return (m == S_primitive || m == S_special ||
                m == S_special_cbv || m == S_void)? S_true: S_false;
}
```

(explode symbol) ⟶ list

```
int z_explode(int n) {
        int     m, y, a;
        char    s[2];

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        m = Car[m];
        if (m == NIL) return NIL;
        if (!symbolic(m)) return error("explode: got non-symbol", m);
        y = alloc(NIL, NIL);
        save(y);
        a = y;
        m = Car[m];
        s[1] = 0;
        while (m != NIL) {
                s[0] = Car[m];
                Car[a] = add_symbol(s, S_void);
                m = Cdr[m];
                if (m != NIL) {
                        Cdr[a] = alloc(NIL, NIL);
                        a = Cdr[a];
                }
        }
        unsave(1);
        return y;
}
```

(implode list) ⟶ symbol

```
int z_implode(int n) {
        int     m, i;
        char    s[SYMBOL_LEN];

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        m = Car[m];
        if (m == NIL) return NIL;
        i = 0;
        while (m != NIL) {
                if (!symbolic(Car[m]))
                        return error("implode: non-symbol in argument",
                                Car[m]);
                if (cdaar(m) != NIL)
                        return error(
                          "implode: input symbol has multiple characters",
                                Car[m]);
                if (i >= SYMBOL_LEN-1)
                        return error("implode: output symbol too long", m);
                s[i] = caaar(m);
                i += 1;
                m = Cdr[m];
        }
        s[i] = 0;
        return add_symbol(s, S_void);
}
```

The following functions deal with recursive lexical environments. They are used by the **`recursive-bind`** function and the **`letrec`** special form.

The `fix_cached_closures()` function fixes recursive bindings created by **`letrec`**. When a recursive function is created using **`letrec`**, **`lambda`** closes over the function name before binding the function to its name. This is demonstrated here using **`let`** instead of **`letrec`**:

```
(closure-form env)
(let ((f (lambda (x) (f x)))) f) => (closure #x #fx ((f . {void})))
```

Whenever a closure is created by **`lambda`**, the lexical environment of that closure is saved on the environment stack (`Env_stack`). When **`letrec`** finishes creating its bindings, it calls `fix_cached_closures()` in order to fix recursive bindings of the above form. To delimit its scope **`letrec`** pushes **`:t`** to the `Env_stack` before starting to process bindings.

When `fix_cached_closures()` is called, the `Env_stack` contains a list of lexical environments like `((f . {void}))` above. The car of the binding stack (`Bind_stack`) contains a list of symbols bound by **`letrec`**.

What `fix_cached_closures()` does now is to traverse each lexical environment in the topmost scope of `Env_stack` and check whether any of its variables is contained in the list on `Bind_stack`. When it finds such a binding, it changes the value associated with that variable in the environment with the value of the symbol on `Bind_stack`. Thereby it changes the above example as follows:

(closure #x #fx ((f . {void}))) —> (closure #x #fx ((f . f_{outer})))

Here f_{outer} refers to the outer binding of f (from `Bind_stack`). Because the outer f binds to the structure containing the fixed environment, a recursive structure is created.

```
void fix_cached_closures(void) {
        int     a, ee, e;

        if (Error_flag || Env_stack == NIL || Env_stack == S_true) return;
        a = Car[Bind_stack];
        while (a != NIL) {
                ee = Env_stack;
                while (ee != NIL && ee != S_true) {
                        e = Car[ee];
                        while (e != NIL) {
                                if (Car[a] == caar(e)) {
                                        cdar(e) = cdar(a);
                                        break;
                                }
                                e = Cdr[e];
                        }
                        ee = Cdr[ee];
                }
                a = Cdr[a];
        }
}
```

Check whether **n** is an association list (an "alist").

```
int is_alist(int n) {
        if (symbolic(n)) return 0;
        while (n != NIL) {
                if (symbolic(Car[n]) || !symbolic(caar(n)))
                        return 0;
                n = Cdr[n];
        }
        return 1;
}
```

The following function is like `fix_cached_closures()`, but instead of fixing cached environments (on the `Env_stack`), it traverses all bindings of a given environment **n** and fixes the bindings of the symbols explicitly specified in **bindings**.

```
void fix_closures_of(int n, int bindings) {
        int     ee, e;
        int     bb, b;

        if (atomic(n)) return;
        if (Car[n] == S_closure) {
                fix_closures_of(caddr(n), bindings);
                ee = cdddr(n);
                if (ee == NIL) return;
                ee = Car[ee];
                while (ee != NIL) {
                        e = Car[ee];
                        bb = bindings;
                        while (bb != NIL) {
                                b = Car[bb];
                                if (Car[b] == Car[e])
                                        Cdr[e] = Cdr[b];
                                bb = Cdr[bb];
                        }
                        ee = Cdr[ee];
                }
                return;
        }
        fix_closures_of(Car[n], bindings);
        fix_closures_of(Cdr[n], bindings);
}
```

Fix all closures of a given environment.

```
void fix_all_closures(int b) {
        int     p;

        p = b;
        while (p != NIL) {
                fix_closures_of(cdar(p), b);
                p = Cdr[p];
        }
}
```

`(recursive-bind alist) --> alist`
Side effect: create cyclic bindings.

```
int z_recursive_bind(int n) {
        int     m, env;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        env = Car[m];
        if (!is_alist(env))
                return error("recursive-bind: bad environment", env);
        fix_all_closures(env);
        return env;
}
```

`(bottom ...) -->` *`undefined`*
Side effect: stop reduction and report an error.

```
int z_bottom(int n) {
        n = alloc(S_bottom, Cdr[n]);
        return error("", n);
}
```

(defined symbol) ⟶ :t | :f

```
int z_defined(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        if (!symbolic(Car[m]))
                return error("defined: got non-symbol", Car[m]);
        return cdar(m) == S_void? S_false: S_true;
}
```

(gc) ⟶ '(*free-nodes peak-usage***)**

```
int z_gc(int n) {
        int     m;
        char    s[20];

        m = Cdr[n];
        if (m != NIL) return wrong_arg_count(n);
        n = alloc(NIL, NIL);
        save(n);
        sprintf(s, "%d", gc());
        Car[n] = explode_string(s);
        Cdr[n] = alloc(NIL, NIL);
        sprintf(s, "%d", Max_atoms_used);
        Max_atoms_used = 0;
        cadr(n) = explode_string(s);
        unsave(1);
        return n;
}
```

(quit) ⟶ *undefined*
Side effect: exit from interpreter.

```
int z_quit(int n) {
        int     m;

        m = Cdr[n];
        if (m != NIL) return wrong_arg_count(n);
        zen_fini();
        exit(0);
}
```

(symbols) ⟶ '(*symbol* **...)**

```
int z_symbols(int n) {
        int     m;

        m = Cdr[n];
        if (m != NIL) return wrong_arg_count(n);
        return Symbols;
}
```

(verify-arrows :t | :f) —> :t | :f
Side effect: turn arrow verification on or off.

```
int z_verify_arrows(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        m = Car[m];
        if (m != S_true && m != S_false)
                return error("verify-arrows: got non truth-value", m);
        Verify_arrows = m == S_true;
        return m;
}
```

If `Car[np[0]]` is a primitive function, run the corresponding primitive operation handler, set `np[0]` to the result of the operation, and return 1. If `Car[np[0]]` is not a primitive function, return 0.

```
int primitive(int *np) {
        int     n, y;
        int     (*op)(int);

        n = np[0];
        y = Car[n];
        if (Error_flag) return 0;
        if (Car[y] == S_primitive) {
                op = Primitives[cadr(y)];
        }
        else {
                return 0;
        }
        n = (*op)(n);
        np[0] = n;
        return 1;
}
```

12.9 special form handlers

A *special form handler* is a function that handles the interpretation of a “special form”. *Special forms* are those forms that constitute the syntax of zenlisp. They are applications of keywords like **lambda**, **define**, and **cond**.

Each special form handler receives four arguments: the special form **n** and three pointers to **int** variables named **pcf**, **pmode**, and **pcbn**. These variables form the *state of the evaluator*.

The handler rewrites the form **n** in a way that is specific to the special form and returns it. The pointers **pcf**, **pmode**, and **pcbn** are used to control what the core of the evaluator does with the rewritten form. **Pmode** is the new mode of the evaluator, **pcbn** controls whether the returned form is evaluated using call-by-name, and **pcf** is the so-called *continue flag*. Setting `pcf[0]` signals the evaluator that the returned form is an expression rather than a value. In this case the evaluation of the form must continue. Hence the name of this flag.

Special form handlers are also responsible for checking the syntax of the forms passed to them.

The `setup_and_or()` function prepares an **`and`** or **`or`** form for reduction.

Things get a bit messy at this point, because the `Bind_stack` is used to store temporary results of control constructs, too. In the following case, it keeps the argument list of **`and`** or **`or`**.

```
int setup_and_or(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL) return wrong_arg_count(n);
        bsave(m);
        return Car[m];
}
```

`(and expression ...) --> form`

```
int z_and(int n, int *pcf, int *pmode, int *pcbn) {
        USE(pcbn);
        if (Cdr[n] == NIL) {
                return S_true;
        }
        else if (cddr(n) == NIL) {
                *pcf = 1;
                return cadr(n);
        }
        else {
                *pcf = 2;
                *pmode = MCONJ;
                return setup_and_or(n);
        }
}
```

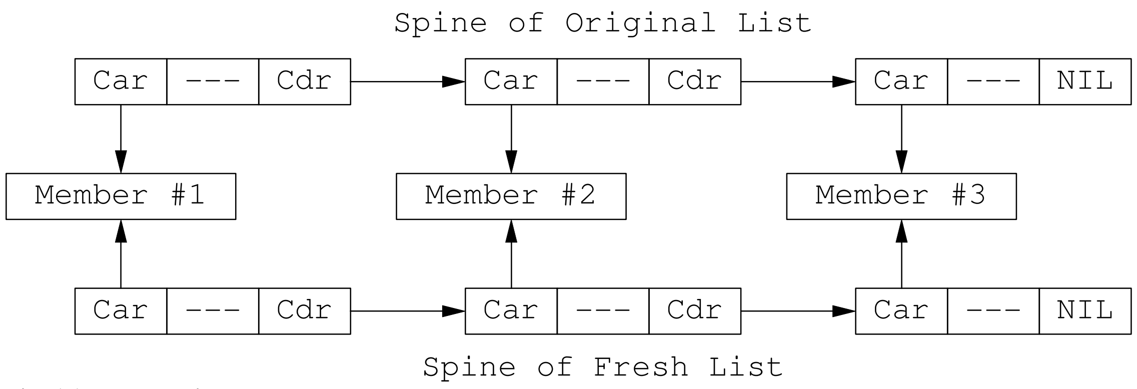

Fig. 16 – shared list

The `flat_copy()` function creates a fresh list that consists of the same objects as the list that was passed to it. Only the nodes that form the *spine* of the list are allocated freshly. Both the original list and the fresh list share the same members. Figure 16 illustrates this principle.

Note that instead of appending a pointer to **()** in above diagram, `NIL` is included in the cdr part of the last box of each list for brevity.

```
int flat_copy(int n, int *lastp) {
        int     a, m, last;

        if (n == NIL) {
                lastp[0] = NIL;
                return NIL;
        }
        m = alloc(NIL, NIL);
        save(m);
        a = m;
        last = m;
        while (n != NIL) {
                Car[a] = Car[n];
                last = a;
                n = Cdr[n];
                if (n != NIL) {
                        Cdr[a] = alloc(NIL, NIL);
                        a = Cdr[a];
                }
        }
        unsave(1);
        lastp[0] = last;
        return m;
}
```

(apply function [expression ...] list) ⟶ form

This handler merely rewrites
(apply function [expression ...] list)
to
(function [expression ...] . list)
and returns it. Note the dot before the *list* argument! When the **apply** handler finishes, the evaluator re-reduces the returned expression.

```
int z_apply(int n, int *pcf, int *pmode, int *pcbn) {
        int     m, p, q, last;
        char    *err1 = "apply: got non-function",
                *err2 = "apply: improper argument list";

        *pcf = 1;
        USE(pmode);
        *pcbn = 1;
        m = Cdr[n];
        if (m == NIL || Cdr[m] == NIL) return wrong_arg_count(n);
        if (atomic(Car[m])) return error(err1, Car[m]);
        p = caar(m);
        if (    p != S_primitive &&
                p != S_special &&
                p != S_special_cbv &&
                p != S_closure
        )
                return error(err1, Car[m]);
        p = Cdr[m];
        USE(last);
        while (p != NIL) {
                if (symbolic(p)) return error(err2, cadr(m));
                last = p;
                p = Cdr[p];
        }
        p = Car[last];
        while (p != NIL) {
                if (symbolic(p)) return error(err2, Car[last]);
                p = Cdr[p];
        }
        if (cddr(m) == NIL) {
                p = cadr(m);
        }
        else {
                p = flat_copy(Cdr[m], &q);
                q = p;
                while (cddr(q) != NIL) q = Cdr[q];
                Cdr[q] = Car[last];
        }
        return alloc(Car[m], p);
}
```

Extract the predicate of the current clause of a **cond** expression. `Car[Bind_stack]` holds the clauses.

```
int cond_get_pred(void) {
        int     e;

        e = caar(Bind_stack);
        if (atomic(e) || atomic(Cdr[e]) || cddr(e) != NIL)
                return error("cond: bad clause", e);
        return Car[e];
}
```

Prepare a **cond** expression for evaluation. Save the clauses on the `Bind_stack` and return the first predicate.

```
int cond_setup(int n) {
        int     m;
        m = Cdr[n];
        if (m == NIL) return wrong_arg_count(n);
        bsave(m);
        return cond_get_pred();
}
```

Evaluate next clause of **cond**. **N** is the value of the current predicate. If **n=:f**, return the predicate of the next clause. If **n≠:f**, return the expression associated with that predicate (the body of the clause). When returning the body of a clause, set the context on `Bind_stack` to **()** to signal that the evaluation of the **cond** expression is complete.

```
int cond_eval_clause(int n) {
        int     e;

        e = Car[Bind_stack];
        if (n == S_false) {
                Car[Bind_stack] = Cdr[e];
                if (Car[Bind_stack] == NIL)
                        return error("cond: no default", NO_EXPR);
                return cond_get_pred();
        }
        else {
                e = cadar(e);
                Car[Bind_stack] = NIL;
                return e;
        }
}
```

(cond (predicate$_1$ expression$_1$)
(predicate$_2$ expression$_2$)
...) ⟶ form

```
int z_cond(int n, int *pcf, int *pmode, int *pcbn) {
        *pcf = 2;
        *pmode = MCOND;
        USE(pcbn);
        return cond_setup(n);
}
```

Check whether **m** is a list of symbols.

```
int is_list_of_symbols(int m) {
        while (m != NIL) {
                if (!symbolic(Car[m])) return 0;
                if (symbolic(Cdr[m])) break;
                m = Cdr[m];
        }
        return 1;
}
```

(define (symbol$_1$ symbol$_2$ ...) expression) ⟶ symbol

This function rewrites function definitions of the above form to
(lambda (symbol$_2$ ...) expression)
evaluates that expression and binds its normal form to **symbol$_1$**.
Side effect: create global binding.

```
int define_function(int n) {
        int     m, y;

        m = Cdr[n];
        if (Car[m] == NIL)
                return error("define: missing function name",
                        Car[m]);
        if (!is_list_of_symbols(Car[m])) return bad_argument_list(Car[m]);
        y = caar(m);
        save(cadr(m));
        Tmp2 = alloc(S_lambda, NIL);
        Cdr[Tmp2] = alloc(cdar(m), NIL);
        cddr(Tmp2) = alloc(cadr(m), NIL);
        cdddr(Tmp2) = alloc(NIL, NIL);
        Cdr[y] = eval(Tmp2);
        Tmp2 = NIL;
        unsave(1);
        return y;
}
```

(define (symbol$_1$ symbol$_2$ ...) expression) ⟶ symbol
(define symbol expression) ⟶ symbol

Evaluate an expression and bind its normal form to a symbol. If the expression is a **lambda** special form, create a closure with an *empty* lexical environment, thereby effectively implementing dynamic scoping.
Side effect: create global binding.

```
int z_define(int n, int *pcf, int *pmode, int *pcbn) {
        int     m, v, y;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        if (Eval_level > 1) {
                error("define: limited to top level", NO_EXPR);
                return NIL;
        }
        m = Cdr[n];
        if (m == NIL || Cdr[m] == NIL || cddr(m) != NIL)
                return wrong_arg_count(n);
        y = Car[m];
        if (!symbolic(y)) return define_function(n);
        v = cadr(m);
        save(v);
        /* If we are binding to a lambda expression, */
        /* add a null environment */
        if (!atomic(v) && Car[v] == S_lambda) {
                if (    Cdr[v] != NIL && cddr(v) != NIL &&
                        cdddr(v) == NIL
                ) {
                        cdddr(v) = alloc(NIL, NIL);
                }
        }
        Cdr[y] = eval(cadr(m));
        unsave(1);
        return y;
}
```

(eval expression) —> form

The `z_eval()` function just returns the expression passed to it for further reduction. Hence **((lambda (x) (x x)) (lambda (x) (eval '(x x))))** reduces in constant space.

```
int z_eval(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;

        *pcf = 1;
        USE(pmode);
        *pcbn = 0;
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        return (Car[m]);
}
```

The following functions deal with closure generation. They are used by the **lambda** special form.

The `is_bound()` function checks whether the symbol **n** is bound in the current context. The context is specified by the variables `Bound_vars`, which contains a (potentially improper) list of variables, and `Car[Lexical_env]`, which holds the lexical environment built so far. `Is_bound()` returns a flag indicating whether the variable is bound.

```
int is_bound(int n) {
        int     b;

        b = Bound_vars;
        while (b != NIL) {
                if (symbolic(b)) {
                        if (n == b) return 1;
                        break;
                }
                if (n == Car[b]) return 1;
                b = Cdr[b];
        }
        b = Car[Lexical_env];
        while (b != NIL) {
                if (caar(b) == n) return 1;
                b = Cdr[b];
        }
        return 0;
}
```

`collect_free_vars()` collects the free variables of a lambda expression. This function expects an empty environment in the car field of `Lexical_env` and a list of bound variables in `Bound_vars`. It returns nothing, but builds the lexical environment in `Car[Lexical_env]`.

The function does not traverse expressions that begin with the keyword **`quote`**. To do so, it is not sufficient to just check for `Car[n] == S_quote`, because doing so would also catch expressions like **`(list quote foo)`**. By checking `caar(n)` instead, it makes sure that **`quote`** actually is in a car position. **Note**: this also prevents `(quote . {internal quote})` from being included, but who wants to re-define **`quote`** anyway?

```
void collect_free_vars(int n) {
        if (n == NIL || (Tag[n] & ATOM_FLAG)) return;
        if (symbolic(n)) {
                if (is_bound(n)) return;
                Car[Lexical_env] = alloc(NIL, Car[Lexical_env]);
                caar(Lexical_env) = alloc(n, Car[n] == Cdr[n]? n: Cdr[n]);
                return;
        }
        if (atomic(Car[n]) || caar(n) != S_quote)
                collect_free_vars(Car[n]);
        collect_free_vars(Cdr[n]);
}
```

The following function creates a lexical environment (a.k.a *lexical context*) for a closure. It is not sufficient to capture the current environment, because the interpreter uses *shallow binding*, where values are stored directly in symbols. `Make_lexical_env()` traverses the function term `term` and collects all free variables contained in it. `Locals` is the argument list of the lambda form to convert to a closure. Local variables are not collected because they are bound by **`lambda`**.

```
int make_lexical_env(int term, int locals) {
        Lexical_env = alloc(NIL, NIL);
        save(Lexical_env);
        Bound_vars = locals;
        collect_free_vars(term);
        unsave(1);
        return Car[Lexical_env];
}
```

Convert a lambda form to a closure:
`(lambda (symbol ...) expression)`
—> `(closure (symbol ...) expression alist)`

Save the environment of the closure on the `Env_stack` so it can be fixed by `fix_cached_closures()`.

```
int make_closure(int n) {
        int     cl, env, args, term;

        if (Error_flag) return NIL;
        args = cadr(n);
        term = caddr(n);
        if (cdddr(n) == NIL) {
                env = make_lexical_env(term, args);
                if (env != NIL) {
                        if (Env_stack != NIL)
                                Env_stack = alloc(env, Env_stack);
                        cl = alloc(env, NIL);
                }
                else {
                        cl = NIL;
                }
        }
        else {
                cl = alloc(cadddr(n), NIL);
        }
        cl = alloc(term, cl);
        cl = alloc(args, cl);
        cl = alloc(S_closure, cl);
        return cl;
}
```

`(lambda (symbol ...) expression) —> {closure ...}`

```
int z_lambda(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;

        m = Cdr[n];
        if (    m == NIL || Cdr[m] == NIL ||
                (cddr(m) != NIL && cdddr(m) != NIL)
        )
                return wrong_arg_count(n);
        if (cddr(m) != NIL && !is_alist(caddr(m)))
                return error("lambda: bad environment",
                        caddr(m));
        if (!symbolic(Car[m]) && !is_list_of_symbols(Car[m]))
                return bad_argument_list(Car[m]);
        return Car[n] == S_closure? n: make_closure(n);
}
```

The `unbind_args()` function is used by the evaluator to restore the call frame of the caller of the current function. It restores the frame pointer (`Frame`), the name of the currently active function (`Function_name`), and the outer values of all symbols bound in the current frame.

```
void unbind_args(void) {
        int     v;

        Frame = unsave(1);
        Function_name = unsave(1);
        v = bunsave(1);
        while (v != NIL) {
                cdar(v) = unsave(1);
                v = Cdr[v];
        }
}
```

The next function sets up a context for the reduction of **`let`** and **`letrec`** special forms. The function saves the complete special form, an additional copy of the environment, an (initially empty) list of symbols to re-bind, and an empty set of inner bindings on `Bind_stack`. It also saves `Env_stack` on `Stack` and creates an empty `Env_stack`. It returns the environment to be processed (the first argument of the special form).

```
int let_setup(int n) {
        int     m;

        m = Cdr[n];
        if (m == NIL || Cdr[m] == NIL || cddr(m) != NIL)
                return wrong_arg_count(n);
        m = Car[m];
        if (symbolic(m))
                return error("let/letrec: bad environment", m);
        bsave(n);        /* save entire LET/LETREC */
        bsave(m);        /* save environment */
        bsave(NIL);      /* list of bindings */
        bsave(NIL);      /* save empty name list */
        save(Env_stack); /* get outer bindings out of the way */
        Env_stack = NIL;
        return m;
}
```

Process one binding of **`let/letrec`**. Add the current binding to the list of new bindings and remove it from the environment of `Bind_stack`. Return the rest of the environment. When this function returns **`()`**, all bindings have been processed.

```
int let_next_binding(int n) {
        int     m, p;

        m = caddr(Bind_stack);  /* rest of environment */
        if (m == NIL) return NIL;
        p = Car[m];
        Tmp2 = n;
        cadr(Bind_stack) = alloc(NIL, cadr(Bind_stack));
        caadr(Bind_stack) = alloc(Car[p], n);
        Tmp2 = NIL;
        caddr(Bind_stack) = Cdr[m];
        return Cdr[m];
}
```

Evaluate one argument of **`let/letrec`**. Fetch one binding from the environment saved on `Bind_stack` and check its syntax. If the syntax is wrong, clean up the context and bail out. If the binding is well-formed, save its variable (car field) in the name list on `Bind_stack` and return the associated expression (cadr field) for reduction.

```
int let_eval_arg(void) {
        int     m, p, v;

        m = caddr(Bind_stack);
        p = Car[m];
        if (    atomic(p) || Cdr[p] == NIL || atomic(Cdr[p]) ||
                cddr(p) != NIL || !symbolic(Car[p])
        ) {
                /* Error, get rid of the partial environment. */
                v = bunsave(1);
                bunsave(3);
                bsave(v);
                Env_stack = unsave(1);
                save(Function_name);
                save(Frame);
                unbind_args();
                return error("let/letrec: bad binding", p);
        }
        Car[Bind_stack] = alloc(Car[p], Car[Bind_stack]);
        return cadr(p);
}
```

Reverse a list in situ (overwriting the original list).

```
int reverse_in_situ(int n) {
        int     this, next, x;

        if (n == NIL) return NIL;
        this = n;
        next = Cdr[n];
        Cdr[this] = NIL;
        while (next != NIL) {
                x = Cdr[next];
                Cdr[next] = this;
                this = next;
                next = x;
        }
        return this;
}
```

Establish bindings of **`let/letrec`**. Save outer values on `Stack`.

```
void let_bind(int env) {
        int     b;

        while (env != NIL) {
                b = Car[env];
                save(cdar(b));          /* Save old value */
                cdar(b) = Cdr[b];       /* Bind new value */
                env = Cdr[env];
        }
}
```

Finish creation of local bindings by **`let/letrec`**. First load the context from `Bind_stack` into local variables and clean up `Bind_stack`. Then perform the actual bindings, saving outer values on `Stack`. Save a list of local symbols on `Bind_stack` so that they can be restored later. If this function processes a **`letrec`** special form (**rec** set), fix cached lexical environments. Finally return the term of the binding construct for further reduction.

This function leaves the same kind of call frame as **`lambda`** on `Bind_stack` and `Stack`.

```
int let_finish(int rec) {
        int     m, v, b, e;

        Tmp2 = alloc(NIL, NIL); /* Create safe storage */
        Cdr[Tmp2] = alloc(NIL, NIL);
        cddr(Tmp2) = alloc(NIL, NIL);
        cdddr(Tmp2) = alloc(NIL, NIL);
        v = bunsave(1);
        b = bunsave(1);         /* bindings */
        m = bunsave(2);         /* drop environment, get full LET/LETREC */
        b = reverse_in_situ(b); /* needed for UNBINDARGS() */
        e = unsave(1);
        Car[Tmp2] = b;
        cadr(Tmp2) = m;
        caddr(Tmp2) = v;
        cdddr(Tmp2) = e;
        let_bind(b);
        bsave(v);
        if (rec) fix_cached_closures();
        Env_stack = e;
        save(Function_name);
        save(Frame);
        Tmp2 = NIL;
        return caddr(m); /* term */
}
```

(let ((symbol expression$_1$) ...) expression$_n$) ⟶ form

```
int z_let(int n, int *pcf, int *pmode, int *pcbn) {
        *pcf = 2;
        *pmode = MBIND;
        USE(pcbn);
        if (let_setup(n) != NIL)
                return let_eval_arg();
        else
                return NIL;
}
```

(letrec ((symbol expression$_1$) ...) expression$_n$) ⟶ form

```
int z_letrec(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;

        *pcf = 2;
        *pmode = MBINR;
        USE(pcbn);
        if (let_setup(n) != NIL)
                m = let_eval_arg();
        else
                m = NIL;
        Env_stack = S_true;
        return m;
}
```

(or expression ...) ⟶ form

```
int z_or(int n, int *pcf, int *pmode, int *pcbn) {
        USE(pcbn);
        if (Cdr[n] == NIL) {
                return S_false;
        }
        else if (cddr(n) == NIL) {
                *pcf = 1;
                return cadr(n);
        }
        else {
                *pcf = 2;
                *pmode = MDISJ;
                return setup_and_or(n);
        }
}
```

(quote form) ⟶ form

```
int z_quote(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        return (Car[m]);
}
```

(closure-form :t | :f) —> :t | :f

```
int z_closure_form(int n, int *pcf, int *pmode, int *pcbn) {
        int             m;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        if (!symbolic(Car[m]))
                return error("closure-form: got non-symbol", Car[m]);
        if (Car[m] == add_symbol("args", S_void))
                Closure_form = 0;
        else if (Car[m] == add_symbol("body", S_void))
                Closure_form = 1;
        else if (Car[m] == add_symbol("env", S_void))
                Closure_form = 2;
        else
                return S_false;
        return Car[m];
}
```

These variables are saved when dumping an image file.

```
int *Image_vars[] = {
        &Closure_form, &Verify_arrows,
        &Symbols, &Freelist, &S_bottom, &S_closure, &S_false,
        &S_lambda, &S_primitive, &S_quote, &S_special,
        &S_special_cbv, &S_true, &S_void, &S_last,
NULL };
```

Write a node pool image to the given file. When the image cannot be created or written successfully, report an error.

```
int dump_image(char *p) {
        int     fd, n, i;
        int     **v;
        char    magic[17];

        fd = open(p, O_CREAT | O_WRONLY, 0644);
        setmode(fd, O_BINARY);
        if (fd < 0) {
                error("cannot create file", NO_EXPR);
                Error.arg = p;
                return -1;
        }
        strcpy(magic, "ZEN_____________");
        magic[7] = sizeof(int);
        magic[8] = VERSION;
        n = 0x12345678;
        memcpy(&magic[10], &n, sizeof(int));
        write(fd, magic, 16);
        n = Pool_size;
        write(fd, &n, sizeof(int));
        v = Image_vars;
        i = 0;
        while (v[i]) {
                write(fd, v[i], sizeof(int));
                i = i+1;
        }
        if (    write(fd, Car, Pool_size*sizeof(int))
                        != Pool_size*sizeof(int) ||
                write(fd, Cdr, Pool_size*sizeof(int))
                        != Pool_size*sizeof(int) ||
                write(fd, Tag, Pool_size) != Pool_size
        ) {
                error("dump failed", NO_EXPR);
                close(fd);
                return -1;
        }
        close(fd);
        return 0;
}
```

(dump-image symbol) ⟶ :t

```
int z_dump_image(int n, int *pcf, int *pmode, int *pcbn) {
        int             m;
        static char     buf[SYMBOL_LEN], *s;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        if (!symbolic(Car[m]))
                return error("dump-image: got non-symbol",
                                Car[m]);
        s = symbol_to_string(Car[m], buf, SYMBOL_LEN);
        if (s) dump_image(s);
        return S_true;
}
```

The following functions are used by the **`load`** special form. `Get_source_dir()` extracts the directory part of a file path into a buffer. When the file path has no directory part, the buffer is filled with `"."`.

```
void get_source_dir(char *path, char *buf) {
        char    *p;
        if (strlen(path) > 256) {
                error("load: path too long", NO_EXPR);
                return;
        }
        strcpy(buf, path);
        p = strrchr(buf, '/');
        if (p == NULL)
                strcpy(buf, ".");
        else
                *p = 0;
}
```

Expand path names beginning with `"~"` by copying the path to a buffer and replacing the tilde in the copy with `$ZENSRC/` (the value of the `ZENSRC` environment variable and a slash). When `ZENSRC` is undefined, simply return the original path.

```
/* Expand leading ~ in path names */
char *expand_path(char *s, char *buf) {
        char    *r, *v;

        if (s[0] == '~')
                r = &s[1];
        else
                return s;
        if ((v = getenv("ZENSRC")) == NULL) return s;
        if (strlen(v) + strlen(r) + 4 >= MAX_PATH_LEN) {
                error("load: path too long", NO_EXPR);
                return s;
        }
        sprintf(buf, "%s/%s", v, r);
        return buf;
}
```

Load a zenlisp source file. Read expressions from the file and pass them to the evaluator. `Load()` uses `Load_level` keeps track of nested **`load`**s. When loading a file from within a file, the same source path will be used for nested **`loads`**.

Note: the approach used here is buggy, because it does not restore the source path when leaving a directory. Feel free to fix this.

```
int load(char *p) {
        FILE    *ofile, *nfile;
        int     r;
        char    *oname;
        char    *arg;
        int     oline;

        arg = p;
        if (Load_level > 0) {
                if (strlen(p) + strlen(Source_dir) + 4 >= MAX_PATH_LEN) {
                        error("load: path too long", NO_EXPR);
                        return -1;
                }
                if (*p != '.' && *p != '/' && *p != '~')
                        sprintf(Current_path, "%s/%s", Source_dir, p);
                else
                        strcpy(Current_path, p);
                p = Current_path;
        }
        p = expand_path(p, Expanded_path);
        get_source_dir(p, Source_dir);
        strcat(p, ".l");
        if ((nfile = fopen(p, "r")) == NULL) {
                error("cannot open source file", NO_EXPR);
                Error.arg = arg;
                return -1;
        }
        Load_level = Load_level + 1;
        /* Save I/O state and redirect */
        r = Rejected;
        ofile = Input;
        Input = nfile;
        oline = Line;
        Line = 1;
        oname = Infile;
        Infile = p;
        read_eval_loop();
        Infile = oname;
        Line = oline;
        /* Restore previous I/O state */
        Rejected = r;
        Input = ofile;
        Load_level = Load_level - 1;
        fclose(nfile);
        if (Paren_level) error("unbalanced parentheses in loaded file",
                                NO_EXPR);
        return 0;
}
```

(load symbol) ⟶ :t

```
int z_load(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;
        char    buf[SYMBOL_LEN+1], *s;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        if (!symbolic(Car[m])) return error("load: got non-symbol", Car[m]);
        s = symbol_to_string(Car[m], buf, SYMBOL_LEN);
        if (s) {
                s = strdup(s);
                if (s == NULL) fatal("load: strdup() failed");
                load(s);
                free(s);
        }
        return S_true;
}
```

(stats expression) ⟶ '(*form reductions allocations collections*)

```
int z_stats(int n, int *pcf, int *pmode, int *pcbn) {
        int     m;
        char    buf[100];

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL || Cdr[m] != NIL) return wrong_arg_count(n);
        reset_counter(&Allocations);
        reset_counter(&Reductions);
        reset_counter(&Collections);
        Stat_flag = 1;
        n = eval(Car[m]);
        Stat_flag = 0;
        n = alloc(n, NIL);
        save(n);
        Cdr[n] = alloc(NIL, NIL);
        cadr(n) = explode_string(counter_to_string(&Reductions, buf));
        cddr(n) = alloc(NIL, NIL);
        caddr(n) = explode_string(counter_to_string(&Allocations, buf));
        cdddr(n) = alloc(NIL, NIL);
        cadddr(n) = explode_string(counter_to_string(&Collections, buf));
        unsave(1);
        return n;
}
```

(trace symbol) ⟶ :t
(trace) ⟶ :t

```
int z_trace(int n, int *pcf, int *pmode, int *pcbn) {
        int             m;
        static char     buf[SYMBOL_LEN], *s;

        USE(pcf);
        USE(pmode);
        USE(pcbn);
        m = Cdr[n];
        if (m == NIL) {
                Traced_fn = NIL;
                return S_true;
        }
        if (Cdr[m] != NIL) return wrong_arg_count(n);
        if (!symbolic(Car[m])) return error("trace: got non-symbol", Car[m]);
        s = symbol_to_string(Car[m], buf, SYMBOL_LEN);
        if (!s) return S_false;
        Traced_fn = find_symbol(s);
        return S_true;
}
```

If `Car[np[0]]` is a keyword, run the corresponding special form handler, set `np[0]` to the result of the operation, and return 1. If `Car[np[0]]` is not a keyword, return 0.

```
int special(int *np, int *pcf, int *pmode, int *pcbn) {
        int     n, y;
        int     (*op)(int, int *, int *, int *);

        n = np[0];
        y = Car[n];
        if (Error_flag) return 0;
        if (Car[y] == S_special || Car[y] == S_special_cbv)
                op = Specials[cadr(y)];
        else if (symbolic(y) &&
                (cadr(y) == S_special ||
                 cadr(y) == S_special_cbv)
        )
                op = Specials[caddr(y)];
        else
                return 0;
        np[0] = (*op)(n, pcf, pmode, pcbn);
        return 1;
}
```

12.10 evaluator

The `bind_args()` function binds the variables of a lambda function (closure) to some actual arguments:
For **i** in **((lambda (v_1 ... v_b) x) a_1 ... a_n)**,
– add v_i to `Car[Bind_stack]` (intially empty);
– save the value of v_i on `Stack`;
– bind v_i to a_i.

Also save and update the function name and call frame pointer.

Because shallow binding is used, the outer value of each variable has to be saved before re-binding the variable and restored when the context of a function ceases to exist. This approach may look inefficient, but it makes variable lookup a single indirection.

```
void bind_args(int n, int name) {
        int     fa,     /* formal arg list */
                aa,     /* actual arg list */
                e;      /* term */
        int     env;    /* optional lexical environment */
        int     p;
        int     at;     /* atomic argument list flag */

        if (Error_flag) return;
        fa = cadar(n);
        at = symbolic(fa);
        aa = Cdr[n];
        p = cddar(n);
        e = Car[p];
        env = Cdr[p] != NIL ? cadr(p): NIL;
        bsave(NIL); /* names */
        while ((fa != NIL && aa != NIL) || at) {
                if (!at) {
                        Car[Bind_stack] = alloc(Car[fa], Car[Bind_stack]);
                        save(cdar(fa));
                        cdar(fa) = Car[aa];
                        fa = Cdr[fa];
                        aa = Cdr[aa];
                }
                if (symbolic(fa)) {
                        Car[Bind_stack] = alloc(fa, Car[Bind_stack]);
                        save(Cdr[fa]);
                        Cdr[fa] = aa;
                        fa = NIL;
                        aa = NIL;
                        break;
                }
        }
        while (env != NIL) {
                p = Car[env];
                Car[Bind_stack] = alloc(Car[p], Car[Bind_stack]);
                save(cdar(p));
                cdar(p) = Cdr[p];
                env = Cdr[env];
        }
        if (fa != NIL || aa != NIL) {
                wrong_arg_count(n);
                n = NIL;
        }
        else {
                n = e;
        }
        save(Function_name);
        Function_name = name;
        save(Frame);
        Frame = Stack;
}
```

Print a call to a function being traced:
`+ (function argument ...)`

```
void print_trace(int n) {
        pr("+ ");
        pr("(");
        Quotedprint = 1;
        print(Traced_fn);
        while (1) {
                n = Cdr[n];
                if (n == NIL) break;
                pr(" ");
                print(Car[n]);
        }
        pr(")"); nl();
}
```

The `eliminate_tail_calls()` function examines the `Mode_stack` to find out whether the caller of the current function in `MBETA` state. If it is, the call to the current function was a tail call. In this case, `eliminate_tail_calls()` removes all **`let`**, **`letrec`**, and **`lambda`** frames of the caller from `Stack` and `Mode_stack`.

```
void eliminate_tail_calls(void) {
        int     m, y;

        m = Car[Mode_stack];
        /* Skip over callee's local frames, if any */
        while (m != NIL && Car[m] == MLETR) {
                m = Cdr[m];
        }
        /* Parent not beta-reducing? Give up. */
        if (m == NIL || Car[m] != MBETA)
                return;
        /* Yes, this is a tail call: */
        /* remove callee's frames. */
        while (1) {
                Tmp2 = unsave(1); /* M */
                unbind_args();
                unsave(1);
                y = munsave();
                save(Tmp2);
                Tmp2 = NIL;
                if (y == MBETA) break;
        }
}
```

The `eval()` function is the heart of the zenlisp interpreter. It reduces an expression to is normal form and returns it. The function is guaranteed to run in constant space when the program evaluated by it runs in constant space.

Because `eval()` is a bit long (multiples pages), commentary text is interspersed in the function. The first block of code saves and resets current interpreter state.

```
int eval(int n) {
        int     m,      /* Result node */
                m2,     /* Root of result lists */
                a;      /* Used to append to result */
        int     mode,   /* Current state */
                cf,     /* Continue flag */
                cbn;    /* Call by name flag */
        int     nm;     /* Name of function to apply */

        Eval_level = Eval_level + 1;
        save(n);
        save(Arg_stack);
        save(Bind_stack);
        save(Car[Mode_stack]);
        save(Stack_bottom);
        Stack_bottom = Stack;
        mode = MATOM;
        cf = 0;
        cbn = 0;
```

In the following loop, **n** holds a source expression (which may have been generated by a special form) and **m** finally holds the corresponding normal form.

```
        while (!Error_flag) {
                if (Stat_flag) count(&Reductions, 1);
                if (n == NIL) {                 /* () -> () */
                        m = NIL;
                        cbn = 0;
                }
                else if (symbolic(n)) {         /* Symbol -> Value */
                        if (cbn) {
                                m = n;
                                cbn = 0;
                        }
                        else {
                                m = Cdr[n] == Car[n]? n: Cdr[n];
                                if (m == S_void) {
                                        error("symbol not bound", n);
                                        break;
                                }
                        }
                }
```

Hack alert! When the **cbn** “flag” is set to 2, this means “do not evaluate this expression at all” rather than just “call by name”. Hence **cbn**==2 may be considered a “stronger” form of **cbn**==1.

```
                else if (Car[n] == S_closure ||
                        Car[n] == S_primitive ||
                        Car[n] == S_special ||
                        Car[n] == S_special_cbv ||
                        cbn == 2
                ) {
                        m = n;
                        cbn = 0;
                }
```

The following branch is used to descend into a list. It saves various values on the stacks for processing by the loop that follows the branch. The saved values are:

- the original source list (on `Stack`);
- the current state (on `Mode_stack`);
- the result list (filled by the subsequent loop, on `Arg_stack`);
- a pointer for appending values to the result list (on `Arg_stack`);
- the remaining members to evaluate (on `Arg_stack`).

When call by value is used, the result list will be initialized with **`(())`** and the append pointer **a** will point to the same form. To append a normal form **`x`**, it is then sufficient to store it in `Car[a]`. Then a new empty list is stored in `Cdr[a]` and **a** advances to the cdr part. So appending elements is an O(1) operation.

When call by name is used, the result list is a copy of the source list and the remaining member list is set to **`()`**.

```
        else {                        /* List (...) and Pair (X.Y) */
                m = Car[n];
                save(n);
                msave(mode);
                if ((symbolic(m) && cadr(m) == S_special) || cbn) {
                        cbn = 0;
                        asave(NIL);
                        asave(NIL);
                        asave(n);       /* Root of result list */
                        n = NIL;
                }
                else {
                        a = alloc(NIL, NIL);
                        asave(a);
                        asave(Cdr[n]);
                        asave(a);       /* Root of result list */
                        n = Car[n];
                }
                mode = MLIST;
                continue;
        }
```

The following loop evaluates the members of a list, performs function applications and reduces special forms. Note that the indentation of the `while` loop is wrong. The loop body spans more than 100 lines. You will find a remainder at the end of the loop.

In MBETA state, all that is left to do is to clean up the context of the calling function and return to the outer list (if any).

```
        while (1) if (mode == MBETA || mode == MLETR) {
                /* Finish BETA reduction */
                unbind_args();
                unsave(1);
                mode = munsave();
        }
```

In MLIST mode, members of a list are reduced to their normal forms.

```
        else if (mode == MLIST) {
                n = cadr(Arg_stack);     /* Next member */
                a = caddr(Arg_stack);    /* Place to append to */
                m2 = aunsave(1);         /* Root of result list */
```

OK, got a complete list, now decide what do do with it.

```
                if (a != NIL) Car[a] = m;
                if (n == NIL) {          /* End of list */
                        m = m2;
                        aunsave(2);      /* Drop N,A */
                        nm = Car[unsave(1)];
                        save(m);         /* Save result */
                        if (Traced_fn == nm) print_trace(m);
                        if (primitive(&m))
                                ;
                        else if (special(&m, &cf, &mode, &cbn))
                                n = m;
                        else if (!atomic(Car[m]) &&
                                caar(m) == S_closure
                        ) {
                                nm = symbolic(nm)? nm: NIL;
                                eliminate_tail_calls();
                                bind_args(m, nm);
                                /* N=E of ((LAMBDA (...) E) ...) */
                                n = caddar(m);
                                cf = 2;
                                mode = MBETA;
                        }
                        else {
                                error("application of non-function",
                                        nm);
                                n = NIL;
                        }
```

Another “flag” hack. **Cf==2** means that the current context cannot be abandoned yet, because the current evaluation is still in progress. This happens only when evaluating terms of lambda functions. In this case, the clean up is performed by setting the mode=`MBETA`.

```
                                if (cf != 2) {
                                        unsave(1);
                                        mode = munsave();
                                }
                                /* Leave the list loop and re-evaluate N */
                                if (cf) break;
                        }
```

End of list not yet reached, insert current member, append new empty slot, and prepare next member for evaluation.

```
                        else {          /* N =/= NIL: Append to list */
                                asave(m2);
                                Cdr[a] = alloc(NIL, NIL);
                                caddr(Arg_stack) = Cdr[a];
                                cadr(Arg_stack) = Cdr[n];
                                if (symbolic(n))
                                        error("improper list in application",
                                                n);
                                n = Car[n];     /* Evaluate next member */
                                break;
                        }
                }
```

This is the place where the binding constructs and control flow constructs are handled. This is still part of the list evaluating loop.

This branch evaluates **`cond`** expressions.

```
                else if (mode == MCOND) {
                        n = cond_eval_clause(m);
                        if (Car[Bind_stack] == NIL) {
                                unsave(1);
                                bunsave(1);
                                mode = munsave();
                        }
                        cf = 1;
                        break;
                }
```

Evaluate **`and`** and **`or`**.

```
                else if (mode == MCONJ || mode == MDISJ) {
                        Car[Bind_stack] = cdar(Bind_stack);
                        if (    (m == S_false && mode == MCONJ) ||
                                (m != S_false && mode == MDISJ) ||
                                Car[Bind_stack] == NIL
                        ) {
                                unsave(1);
                                bunsave(1);
                                mode = munsave();
                                n = m;
                                cbn = 2;
                        }
                        else if (cdar(Bind_stack) == NIL) {
                                n = caar(Bind_stack);
                                unsave(1);
                                bunsave(1);
                                mode = munsave();
                        }
                        else {
                                n = caar(Bind_stack);
                        }
                        cf = 1;
                        break;
                }
```

Evaluate **`let`** and **`letrec`**.

```
                else if (mode == MBIND || mode == MBINR) {
                        if (let_next_binding(m) == NIL) {
                                n = let_finish(mode == MBINR);
                                mode = MLETR;
                        }
                        else {
                                n = let_eval_arg();
                        }
                        cf = 1;
                        break;
                }
```

If the expression to evaluate was an atom, there is nothing left to do.

```
                else {  /* Atom */
                        break;
                }
```

The list evaluating loop ends above.

```
                if (cf) {       /* Continue evaluation if requested */
                        cf = 0;
                        continue;
                }
                if (Stack == Stack_bottom) break;
        }
```

Restore the previous state of the interpreter.

```
        while (Stack != Stack_bottom) unsave(1);
        Stack_bottom = unsave(1);
        Car[Mode_stack] = unsave(1);
        Bind_stack = unsave(1);
        Arg_stack = unsave(1);
        unsave(1);
        Eval_level = Eval_level - 1;
        return m;
}
```

12.11 printer

The helper functions that precede `print()` all work in the same way. They check whether their argument has a specific property and if it has that property, they print the argument and return 1. Otherwise they return 0.

Print **`(quote x)`** as **`'x`**.

```
int print_quoted_form(int n, int dot) {
        if (    Car[n] == S_quote &&
                Cdr[n] != NIL &&
                cddr(n) == NIL
        ) {
                if (dot) pr(" . ");
                n = cadr(n);
                if (n != S_true && n != S_false) pr("'");
                print(n);
                return 1;
        }
        return 0;
}
```

Print lists of single-character symbols as condensed lists.

```
int print_condensed_list(int n, int dot) {
        int     m;
        char    s[2];

        m = n;
        if (m == NIL) return 0;
        while (m != NIL) {
                if (!symbolic(Car[m])) return 0;
                if (cdaar(m) != NIL) return 0;
                m = Cdr[m];
        }
        if (dot) pr(" . ");
        pr("#");
        m = n;
        s[1] = 0;
        while (m != NIL) {
                s[0] = caaar(m);
                pr(s);
                m = Cdr[m];
        }
        return 1;
}
```

Print closures. The `Closure_form` variable determines the amount of information to print. Note that closures are ambiguous when no environment is printed. Hence this function prints **`{closure ...}`** instead of **`(closure ...)`** when `Closure_form` is less than 2.

```
int print_closure(int n, int dot) {
        if (    Car[n] == S_closure &&
                !atomic(Cdr[n]) &&
                !atomic(cddr(n))
        ) {
                Quotedprint = 1;
                if (dot) pr(" . ");
                pr(Closure_form==2? "(closure ": "{closure ");
                print(cadr(n));
                if (Closure_form > 0) {
                        pr(" ");
                        print(caddr(n));
                        if (Closure_form > 1 && cdddr(n) != NIL) {
                                pr(" ");
                                print(cadddr(n));
                        }
                }
                pr(Closure_form==2? ")": "}");
                return 1;
        }
        return 0;
}
```

Print primitive function handlers and special form handlers.

```
int print_primitive(int n, int dot) {
        if (    Car[n] != S_primitive &&
                Car[n] != S_special &&
                Car[n] != S_special_cbv
        )
                return 0;
        if (dot) pr(" . ");
        pr("{internal ");
        Quotedprint = 1;
        print(cddr(n));
        pr("}");
        return 1;
}
```

This is the zenlisp printer interface. It converts the internal node representation of a form into its external (human-readable) form and emits it.

```
void print(int n) {
        char    s[SYMBOL_LEN+1];
        int     i;

        if (n == NIL) {
                pr("()");
        }
        else if (n == S_void) {
                pr("{void}");
        }
        else if (Tag[n] & ATOM_FLAG) {
                /* Characters are limited to the symbol table */
                pr("{unprintable form}");
        }
        else if (symbolic(n)) {
                if (!Quotedprint && n != S_true && n != S_false) {
                        pr("'");
                        Quotedprint = 1;
                }
                i = 0;          /* Symbol */
                n = Car[n];
                while (n != NIL) {
                        s[i] = Car[n];
                        if (i > SYMBOL_LEN-2) break;
                        i += 1;
                        n = Cdr[n];
                }
                s[i] = 0;
                pr(s);
        }
        else {  /* List */
                if (print_closure(n, 0)) return;
                if (print_primitive(n, 0)) return;
                if (!Quotedprint) {
                        pr("'");
                        Quotedprint = 1;
                }
                if (print_quoted_form(n, 0)) return;
                if (print_condensed_list(n, 0)) return;
                pr("(");
                while (n != NIL) {
                        print(Car[n]);
                        n = Cdr[n];
                        if (symbolic(n) || n == S_void) {
                                pr(" . ");
                                print(n);
                                n = NIL;
                        }
                        if (print_closure(n, 1)) break;
                        if (print_primitive(n, 1)) break;
                        if (print_quoted_form(n, 1)) break;
                        if (n != NIL) pr(" ");
                }
                pr(")");
        }
}
```

12.12 initialization

Reset the state of the interpreter: clear the stacks and debugging variables and reset the level counters.

```
void reset_state(void) {
        Stack = NIL;
        Arg_stack = NIL;
        Bind_stack = NIL;
        Env_stack = NIL;
        Frame = NIL;
        Function_name = NIL;
        Eval_level = 0;
        Paren_level = 0;
}
```

First stage of interpreter initialization. Initialize miscellaneous variables, clear the free list, connect the input/output streams.

```
void init1() {
        /* Misc. variables */
        reset_state();
        Mode_stack = NIL;
        Error_flag = 0;
        Error.arg = NULL;
        Fatal_flag = 0;
        Symbols = NIL;
        Safe_symbols = NIL;
        Tmp_car = NIL;
        Tmp_cdr = NIL;
        Tmp = NIL;
        Tmp2 = NIL;
        Load_level = 0;
        Traced_fn = NIL;
        Max_atoms_used = 0;
        Max_trace = 10;
        Stat_flag = 0;
        Closure_form = 0;
        Verify_arrows = 0;
        Line = 1;
        /* Initialize Freelist */
        Freelist = NIL;
        /* Clear input buffer */
        Infile = NULL;
        Source_dir[0] = 0;
        Input = stdin;
        Output = stdout;
        Rejected = EOT;
}
```

Second stage of interpreter initialization: build free list, create built-in symbols.

```
void init2(void) {
        /*
         * Tags (especially 'primitive and 'special*)
         * must be defined before the primitives.
         * First GC will be triggered HERE
         */
        S_void = add_symbol("{void}", 0);
        S_special = add_symbol("{special}", 0);
        S_special_cbv = add_symbol("{special/cbv}", 0);
        S_primitive = add_symbol("{primitive}", 0);
        S_closure = add_symbol("closure", 0);
        add_primitive("atom", P_ATOM);
        add_special("and", SF_AND, 0);
        add_special("apply", SF_APPLY, 1);
        S_bottom = add_primitive("bottom", P_BOTTOM);
        add_primitive("car", P_CAR);
        add_primitive("cdr", P_CDR);
        add_special("closure-form", SF_CLOSURE_FORM, 0);
        add_special("cond", SF_COND, 0);
        add_primitive("cons", P_CONS);
        add_special("define", SF_DEFINE, 0);
        add_primitive("defined", P_DEFINED);
        add_special("dump-image", SF_DUMP_IMAGE, 0);
        add_special("eval", SF_EVAL, 1);
        add_primitive("eq", P_EQ);
        add_primitive("explode", P_EXPLODE);
        S_false = add_symbol(":f", 0);
        add_primitive("gc", P_GC);
        add_primitive("implode", P_IMPLODE);
        S_lambda = add_special("lambda", SF_LAMBDA, 0);
        add_special("let", SF_LET, 0);
        add_special("letrec", SF_LETREC, 0);
        add_special("load", SF_LOAD, 0);
        add_special("or", SF_OR, 0);
        add_primitive("quit", P_QUIT);
        S_quote = add_special("quote", SF_QUOTE, 0);
        add_primitive("recursive-bind", P_RECURSIVE_BIND);
        add_special("stats", SF_STATS, 0);
        add_primitive("symbols", P_SYMBOLS);
        S_true = add_symbol(":t", 0);
        add_symbol("t", S_true);
        add_special("trace", SF_TRACE, 0);
        add_primitive("verify-arrows", P_VERIFY_ARROWS);
        S_last = add_symbol("**", 0);
        Mode_stack = alloc(NIL, NIL);
        Primitives[P_ATOM] = &z_atom;
        Primitives[P_BOTTOM] = &z_bottom;
        Primitives[P_CAR] = &z_car;
        Primitives[P_CDR] = &z_cdr;
        Primitives[P_CONS] = &z_cons;
        Primitives[P_DEFINED] = &z_defined;
        Primitives[P_EQ] = &z_eq;
        Primitives[P_EXPLODE] = &z_explode;
        Primitives[P_GC] = &z_gc;
        Primitives[P_IMPLODE] = &z_implode;
        Primitives[P_QUIT] = &z_quit;
        Primitives[P_RECURSIVE_BIND] = &z_recursive_bind;
        Primitives[P_SYMBOLS] = &z_symbols;
        Primitives[P_VERIFY_ARROWS] = &z_verify_arrows;
        Specials[SF_AND] = &z_and;
        Specials[SF_APPLY] = &z_apply;
        Specials[SF_CLOSURE_FORM] = &z_closure_form;
        Specials[SF_COND] = &z_cond;
        Specials[SF_DEFINE] = &z_define;
        Specials[SF_DUMP_IMAGE] = &z_dump_image;
        Specials[SF_EVAL] = &z_eval;
        Specials[SF_LAMBDA] = &z_lambda;
        Specials[SF_LET] = &z_let;
        Specials[SF_LETREC] = &z_letrec;
        Specials[SF_LOAD] = &z_load;
        Specials[SF_OR] = &z_or;
        Specials[SF_QUOTE] = &z_quote;
        Specials[SF_STATS] = &z_stats;
        Specials[SF_TRACE] = &z_trace;
}
```

Clear **`stats`** counters.

```
void clear_stats(void) {
        reset_counter(&Reductions);
        reset_counter(&Allocations);
        reset_counter(&Collections);
}
```

12.13 interpreter interface

Load a node pool image from a given file. Return zero upon success and a non-zero value in case of an error.

```
int zen_load_image(char *p) {
        int     fd, n, i;
        char    buf[17];
        int     **v;
        int     bad = 0;
        int     inodes;
        fd = open(p, O_RDONLY);
        setmode(fd, O_BINARY);
        if (fd < 0) {
                error("cannot open image", NO_EXPR);
                Error.arg = p;
                return -1;
        }
        memset(Tag, 0, Pool_size);
        read(fd, buf, 16);
        if (memcmp(buf, "ZEN____", 7)) {
                error("bad image (magic match failed)", NO_EXPR);
                bad = 1;
        }
        if (buf[7] != sizeof(int)) {
                error("bad image (wrong cell size)", NO_EXPR);
                bad = 1;
        }
        if (buf[8] != VERSION) {
                error("bad image (wrong version)", NO_EXPR);
                bad = 1;
        }
        memcpy(&n, &buf[10], sizeof(int));
        if (n != 0x12345678) {
                error("bad image (wrong architecture)", NO_EXPR);
                bad = 1;
        }
        read(fd, &inodes, sizeof(int));
        if (inodes > Pool_size) {
                error("bad image (too many nodes)", NO_EXPR);
                bad = 1;
        }
        v = Image_vars;
        i = 0;
        while (v[i]) {
                read(fd, v[i], sizeof(int));
                i = i+1;
        }
        if (    !bad &&
                (read(fd, Car, inodes*sizeof(int)) != inodes*sizeof(int) ||
                 read(fd, Cdr, inodes*sizeof(int)) != inodes*sizeof(int) ||
                 read(fd, Tag, inodes) != inodes)
        ) {
                error("bad image (bad file size)", NO_EXPR);
                bad = 1;
        }
        close(fd);
        if (bad) Error.arg = p;
        return Error_flag;
}
```

Main initialization. Allocate node pool, clear tags, initialize variables.

```
int zen_init(int nodes, int vgc) {
        Pool_size = nodes? nodes: DEFAULT_NODES;
        Verbose_GC = vgc;
        if (Pool_size < MINIMUM_NODES) return -1;
        if (    (Car = (int *) malloc(Pool_size * sizeof(int))) == NULL ||
                (Cdr = (int *) malloc(Pool_size * sizeof(int))) == NULL ||
                (Tag = (char *) malloc(Pool_size)) == NULL
        ) {
                if (Car) free(Car);
                if (Cdr) free(Cdr);
                if (Tag) free(Tag);
                Car = Cdr = NULL;
                Tag = NULL;
                return -1;
        }
        memset(Tag, 0, Pool_size);
        init1();
        init2();
        return 0;
}
```

De-allocate the node pools.

```
void zen_fini() {
        if (Car) free(Car);
        if (Cdr) free(Cdr);
        if (Tag) free(Tag);
        Car = Cdr = NULL;
        Tag = NULL;
}
```

Stop the interpreter (after receiving an “interrupt” signal (`SIGINT`) from the user).

```
void zen_stop(void) {
        error("interrupted", NO_EXPR);
}
```

I/O interface.

```
void zen_print(int n) {
        Quotedprint = 0;
        print(n);
}

int zen_read(void) {
        Paren_level = 0;
        return zread();
}
```

Create a copy of the symbol table. This copy will be kept in a safe location (`Safe_symbols`), so it can be used to restore the symbol table in case something goes totally wrong.

```
int copy_bindings(void) {
        int     y, p, ny, q;

        p = alloc(NIL, NIL);
        save(p);
        ny = p;
        q = NIL;
        y = Symbols;
        while (y != NIL) {
                Car[p] = alloc(Car[y], cdar(y));
                y = Cdr[y];
                Cdr[p] = alloc(NIL, NIL);
                q = p;
                p = Cdr[p];
        }
        if (q != NIL) Cdr[q] = NIL;
        unsave(1);
        return Car[ny] == NIL? NIL: ny;
}
```

Restore bindings saved by `copy_bindings()`.

```
void restore_bindings(int values) {
        int     b;

        while (values != NIL) {
                b = Car[values];
                cdar(b) = Cdr[b];
                values = Cdr[values];
        }
}
```

Safely reduce an expression to its normal form. Bail out gracefully in case of an error.

```
int zen_eval(int n) {
        save(n);
        Safe_symbols = copy_bindings();
        if (Stat_flag) clear_stats();
        n = eval(n, 0);
        unsave(1);
        if (!Error_flag) {
                Cdr[S_last] = n;
                if (Stack != NIL)
                        fatal("eval(): unbalanced stack");
        }
        else {
                restore_bindings(Safe_symbols);
        }
        reset_state();
        while (Car[Mode_stack] != NIL) munsave();
        return n;
}
```

The obligatory license text.

```
char **zen_license() {
        static char     *license_text[] = {
"",
"zenlisp -- An interpreter for symbolic LISP",
"By Nils M Holm, 2007, 2008",
"",
"Don't worry, be happy.",
"",
"THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND",
"ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE",
"IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE",
"ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE",
"FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL",
"DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS",
"OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)",
"HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT",
"LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY",
"OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF",
"SUCH DAMAGE.",
"",
        NULL};
        return license_text;
}
```

This is a simple, internal read-eval loop (without print). It is used for loading programs.

```
void read_eval_loop(void) {
        int     n, evl;

        Error_flag = 0;
        evl = Eval_level;
        Eval_level = 0;
        while(!Error_flag) {
                n = zen_read();
                if (n == EOT) break;
                n = eval(n, 0);
        }
        Eval_level = evl;
}
```

12.14 interpreter shell

Here begins the user interface of the zenlisp interpreter. There are only very few connections between the code above and below. Both parts could be placed in different files with few modifications.

This header is required for handling keyboard interrupts:

```
#include <signal.h>
```

`Image` holds the name of the node pool image to load. `Nodes` is the size of the node pool to allocate. `Batch` is a flag indicating whether the interpreter shall run in batch mode. When `GC_stats` is set to 1, the interpreter will print some information after each garbage collection.

In *batch mode*, the interpreter will

– not print a banner or **=>** operators;
– exit immediately after reporting an error;
– exit immediately when catching `SIGINT`.

```
char    Image[MAX_PATH_LEN];
int     Nodes;
int     Batch;
int     GC_stats;

void usage(void) {
        fprintf(stderr,
                "Usage: zl [-L] [-bgi] [-n nodes] [image]\n");
}
```

Retrieve the (numeric) value associated with a command line option. See `get_options()` for the meanings of **pi**, **pj**, **pk**. Return the retrieved value. Values are normally specified in ones, but a suffix of "K" or "M" may be used to specify "kilos" (1024's) or "megas" (1024^2's) respectively.

```
int get_opt_val(int argc, char **argv, int *pi, int *pj, int *pk) {
        int     n, c;

        if (++(*pi) >= argc) {
                usage();
                exit(1);
        }
        n = atoi(argv[*pi]);
        c = argv[*pi][strlen(argv[*pi])-1];
        switch (c) {
        case 'K':       n = n * 1024; break;
        case 'M':       n = n * 1024 * 1024; break;
        }
        *pj = *pk = 0;
        return n;
}

void help(void) {
        fputc('\n', stderr);
        usage();
        fprintf(stderr,
                "\n"
                "-b    batch mode (quiet, exit on first error)\n"
                "-g    report number of free nodes after each GC\n"
                "-i    init mode (do not load any image)\n"
                "-n #  number of nodes to allocate (default: %dK)\n"
                "-L    print license and exit\n"
                "\n"
                "default image: %s\n\n",
                DEFAULT_NODES/1024, DEFAULT_IMAGE);
}

void print_license(void) {
        char    **s;

        s = zen_license();
        while (*s) {
                printf("%s\n", *s);
                s++;
        }
        exit(0);
}
```

Parse the command line options passed to the interpreter shell. Set default values in case no options are given. The variables **i** (current option), **j** (current character of option string), and **k** (length of current option string) are passed to `get_opt_val()` as pointers in order to extract argument values of options.

```
void get_options(int argc, char **argv) {
        char    *a;
        int     i, j, k;
        int     v;

        strncpy(Image, DEFAULT_IMAGE, strlen(DEFAULT_IMAGE));
        Image[MAX_PATH_LEN-1] = 0;
        Nodes = DEFAULT_NODES;
        GC_stats = 0;
        Batch = 0;
        v = 0;
        i = 1;
        while (i < argc) {
                a = argv[i];
                if (a[0] != '-') break;
                k = strlen(a);
                for (j=1; j<k; j++) {
                        switch (a[j]) {
                        case 'b':
                                Batch = 1;
                                break;
                        case 'n':
                                Nodes = get_opt_val(argc, argv, &i, &j, &k);
                                break;
                        case 'g':
                                GC_stats = 1;
                                break;
                        case 'i':
                                Image[0] = 0;
                                break;
                        case 'L':
                                print_license();
                                break;
                        case '?':
                        case 'h':
                                help();
                                exit(1);
                                break;
                        default:
                                usage();
                                exit(1);
                        }
                }
                i = i+1;
        }
        if (i < argc) {
                strncpy(Image, a, strlen(a)+1);
                Image[MAX_PATH_LEN-1] = 0;
        }
        if (Nodes < MINIMUM_NODES) {
                fprintf(stderr, "zenlisp: minimal pool size is %d\n",
                        MINIMUM_NODES);
                exit(1);
        }
}
```

`SIGINT` handler.

```
void catch_int(int sig) {
        USE(sig);
        zen_stop();
        signal(SIGINT, catch_int);
}
```

The `repl()` function implements the main interpreter loop, the so-called REPL (**r**ead-**e**val-**p**rint **l**oop). As is name suggests, it reads an expression from the input stream, evaluates it, prints the resulting normal form (if any) and finally loops. The REPL exits when it receives an EOT character (or after reporting an error in batch mode).

```
void repl(void) {
        int     n;

        while(1) {
                Error_flag = 0;
                n = zen_read();
                if (n == EOT) return;
                if (Error_flag) {
                        zen_print_error();
                        if (Batch) exit(1);
                        continue;
                }
                n = zen_eval(n);
                if (Error_flag) {
                        zen_print_error();
                        if (Batch) exit(1);
                }
                else {
                        if (!Batch) pr("=> ");
                        zen_print(n);
                        nl();
                }
        }
}

void init(void) {
        if (zen_init(Nodes, GC_stats)) {
                fprintf(stderr, "zenlisp init failed (memory problem)\n");
                exit(1);
        }
}
```

Ready to lift off...

In case you wonder why options are checked twice: the first pass sets pre-initialization options, the second one post-initialization options.

```
int main(int argc, char **argv) {
        get_options(argc, argv);
        init();
        get_options(argc, argv);
        if (!Batch) {
                pr("zenlisp ");
                pr(RELEASE);
                pr(" by Nils M Holm");
                nl();
        }
        if (Image[0]) {
                if (zen_load_image(Image)) {
                        zen_print_error();
                        if (Batch) exit(1);
                        zen_fini();
                        init();
                        get_options(argc, argv);
                }
        }
        else if (!Batch) {
                pr("Warning: no image loaded");
                nl();
        }
        signal(SIGINT, catch_int);
        repl();
        zen_fini();
        return 0;
}
```

13. lisp part

13.1 base library

This part describes the LISP functions contained in the default image of the zenlisp interpreter. They are contained in the file `base.l`. The functions defined here have been discussed in detail in the first part of this book.

Each function definition is preceded by a prototype. Prototypes provide additional, semi-formal information about a function, but they are not part of the zenlisp language. See page 222 for an explanation of function prototypes.

Names of functions that are part of the zenlisp language print in boldface characters in their definitions. Function names not printed in boldface characters are internal to their packages and should never be used in user-level zenlisp code.

```
; zenlisp base functions
; By Nils M Holm, 2007, 2008
; Feel free to copy, share, and modify this code.
; See the file LICENSE for details.

(define base :t)
```

(null form) ⟶ :t | :f

```
(define (null x) (eq x ()))
```

(id form) ⟶ form

```
(define (id x) x)
```

(list form ...) ⟶ list

```
(define (list . x) x)
```

(not form) ⟶ :t | :f

```
(define (not a) (eq a :f))
```

(neq form$_1$ form$_2$) ⟶ :t | :f

```
(define (neq x y) (eq (eq x y) :f))
```

(caar pair) ⟶ form
...
(cddddr pair) ⟶ form

```
(define (caaaar x) (car (car (car (car x)))))
(define (caaadr x) (car (car (car (cdr x)))))
```

```
(define (caadar x) (car (car (cdr (car x)))))
(define (caaddr x) (car (car (cdr (cdr x)))))
(define (cadaar x) (car (cdr (car (car x)))))
(define (cadadr x) (car (cdr (car (cdr x)))))
(define (caddar x) (car (cdr (cdr (car x)))))
(define (cadddr x) (car (cdr (cdr (cdr x)))))
(define (cdaaar x) (cdr (car (car (car x)))))
(define (cdaadr x) (cdr (car (car (cdr x)))))
(define (cdadar x) (cdr (car (cdr (car x)))))
(define (cdaddr x) (cdr (car (cdr (cdr x)))))
(define (cddaar x) (cdr (cdr (car (car x)))))
(define (cddadr x) (cdr (cdr (car (cdr x)))))
(define (cdddar x) (cdr (cdr (cdr (car x)))))
(define (cddddr x) (cdr (cdr (cdr (cdr x)))))

(define (caaar x) (car (car (car x))))
(define (caadr x) (car (car (cdr x))))
(define (cadar x) (car (cdr (car x))))
(define (caddr x) (car (cdr (cdr x))))
(define (cdaar x) (cdr (car (car x))))
(define (cdadr x) (cdr (car (cdr x))))
(define (cddar x) (cdr (cdr (car x))))
(define (cdddr x) (cdr (cdr (cdr x))))

(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))
```

(fold function form list) ⟶ form

```
(define (fold f x a)
  (letrec
    ((fold2
       (lambda (a res)
         (cond ((null a) res)
               (t (fold2 (cdr a)
                         (f res (car a))))))))
    (fold2 a x)))
```

(fold-r function form list) ⟶ form

```
(define (fold-r f x a)
  (letrec
    ((fold2
       (lambda (a)
         (cond ((null a) x)
               (t (f (car a)
                     (fold2 (cdr a))))))))
    (fold2 a)))
```

(reverse list) ⟶ list

```
(define (reverse a)
  (letrec
    ((reverse2
       (lambda (a b)
         (cond ((null a) b)
               (t (reverse2 (cdr a)
                            (cons (car a) b)))))))
    (reverse2 a ())))
```

(append list ...) ⟶ list
(append atom) ⟶ atom
(append list$_1$ list$_2$ ... atom) ⟶ dotted list

```
(define (append . a)
  (letrec
    ((append2
       (lambda (a b)
         (cond ((null a) b)
               (t (append2 (cdr a) (cons (car a) b)))))))
    (fold (lambda (a b) (append2 (reverse a) b))
          ()
          a)))
```

(equal form$_1$ form$_2$) ⟶ :t | :f

The first clause of **equal** returns truth immediately when comparing identical structures (like shared tails of lists). This is a performance hack.

```
(define (equal a b)
  (cond ((eq a b) :t)
        ((or (atom a) (atom b))
          (eq a b))
        (t (and (equal (car a) (car b))
                (equal (cdr a) (cdr b))))))
```

(assoc form alist) ⟶ pair | :f

```
(define (assoc x a)
  (cond ((null a) :f)
        ((equal (caar a) x) (car a))
        (t (assoc x (cdr a)))))
```

(assq atom alist) ⟶ pair | :f

```
(define (assq x a)
  (cond ((null a) :f)
        ((eq (caar a) x) (car a))
        (t (assq x (cdr a)))))
```

(listp form) ⟶ :t | :f

```
(define (listp x)
  (or (null x)
      (and (not (atom x))
           (listp (cdr x)))))
```

(map function $list_1$ $list_2$ ...) ⟶ list

```
(define (map f . a)
  (letrec
    ((map-car
       (lambda (f a r)
         (cond ((null a) (reverse r))
               (t (map-car f (cdr a) (cons (f (car a)) r))))))
     (car-of
       (lambda (a)
         (map-car car a ())))
     (cdr-of
       (lambda (a)
         (map-car cdr a ())))
     (any-null
       (lambda (a)
         (apply or (map-car null a ()))))
     (map2
       (lambda (a b)
         (cond ((any-null a) (reverse b))
               (t (map2 (cdr-of a)
                        (cons (apply f (car-of a)) b)))))))
    (cond ((null a) (bottom '(too few arguments to map)))
          (t (map2 a ())))))
```

(member form list) ⟶ form | :f

```
(define (member x a)
  (cond ((null a) :f)
        ((equal (car a) x) a)
        (t (member x (cdr a)))))
```

(memq atom list) ⟶ form | :f

```
(define (memq x a)
  (cond ((null a) :f)
        ((eq (car a) x) a)
        (t (memq x (cdr a)))))
```

(require symbol) ⟶ :t | :f

```
(define (require x)
  (letrec
    ((require2
       (lambda (sym file)
         (cond ((defined sym) :f)
               (t (apply load (list file)))))))
    (let ((xx (explode x)))
      (cond ((eq (car xx) '~)
              (require2 (implode (cdr xx)) x))
            (t (require2 x x))))))
```

13.2 iterator package

The **iter** package defines iterators for arithmetic function and predicates. These functions have been discussed in great detail in the first part of this book [pages 45 and 53].

The iterator package is contained in the file `iter.l`.

```
; zenlisp iterators
; By Nils M Holm, 2007, 2008
; Feel free to copy, share, and modify this code.
; See the file LICENSE for details.

(define iter :t)
```

(arithmetic-iterator function$_1$ function$_2$ number) ⟶ function

```
(define (arithmetic-iterator conv fn neutral)
  (lambda x
    (cond ((null x) neutral)
          (t (fold (lambda (a b)
                     (fn (conv a) (conv b)))
                   (car x)
                   (cdr x))))))
```

(predicate-iterator function$_1$ function$_2$) ⟶ function

```
(define (predicate-iterator conv fn)
  (let ((fail (cons 'fail ())))
    (let ((comp (lambda (a b)
                  (cond ((eq a fail) fail)
                        ((fn (conv a) (conv b)) b)
                        (t fail)))))
      (lambda (first . rest)
        (cond ((null rest) (bottom '(too few arguments)))
              (t (neq (fold comp first rest) fail)))))))
```

13.3 natural math functions

The **nmath** package implements natural number arithmetics on top of the symbols `'0`...`'9`. Each natural number is a list of these symbols. This is why numbers can be written in condensed form, e.g. **'#31415** instead of **'(3 1 4 1 5)**. In fact, condensed lists were originally invented in order to provide a more convenient notation for numbers. This is also the reason why the "number" sign was chosen to introduce condensed lists.

The **nmath** package forms the foundation of the numeric tower of zenlisp as depicted in figure 17. Each layer of the numeric towers builds on top of the ‘‘lower’’ levels that implement more primitive types.

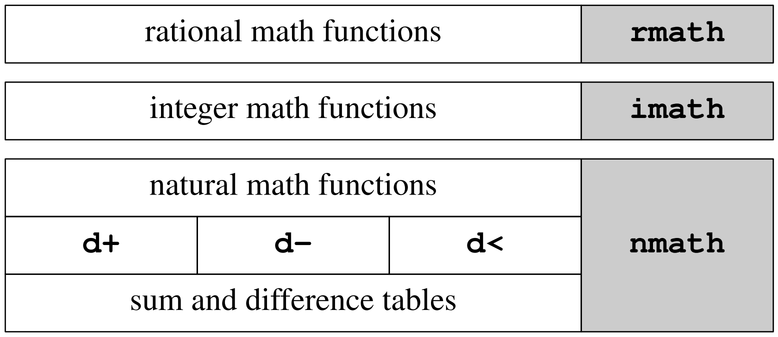

Fig. 17 – numeric tower

Nmath itself consists of three layers implementing arithmetic tables, functions that use these tables to implement single-digit arithmetics and, finally, the natural math functions themselves.

Each of the math packages can be loaded individually, but loading **imath** will include **nmath**, and loading **rmath** will load the complete numeric tower.

While the more primitive math packages lack some functionality that the more complex packages provide, they are often preferable because they are much more efficient when performing the same task.

For example, computing 2^{100} using natural arithmetics is *much* faster than using the corresponding rational math function. The natural and integer versions, on the other hand, do not accept negative exponents. Hence

Math packages should be chosen carefully.

The natural math package is contained in the file nmath.l.

```
; zenlisp natural math functions
; By Nils M Holm, 2007
; Feel free to copy, share, and modify this code.
; See the file LICENSE for details.
```

Nmath requires **base**, but **require** is defined in **base**, so it cannot be used here:

```
(cond ((defined 'base) :f)
      (t (load base)))

(define nmath :t)
```

First define the digit symbols as constants, so you can write `0` instead of `'0`.

```
(define 0 '0)
(define 1 '1)
(define 2 '2)
(define 3 '3)
(define 4 '4)
(define 5 '5)
(define 6 '6)
(define 7 '7)
(define 8 '8)
(define 9 '9)

(define *digits* '#0123456789)
```

(digitp form) ⟶ :t | :f

```
(define (digitp x) (and (memq x *digits*) :t))
```

The **succ** and **pred** functions compute the successor and predecessor of a digit. Both of them return **:f** when an overflow or underflow occurs. The functions are not used much in the code. **Pred** is used only a few times and **succ** is not used at all (it is kept for reasons of symmetry, though). Both functions were involved in the implementation of more complex numeric functions in earlier versions (ArrowLISP), but zenlisp uses table-driven arithmetics instead. See below for details.

(succ symbol) ⟶ symbol | :f

```
(define (succ x)
  (cond ((eq x 0) 1)
        ((eq x 1) 2)
        ((eq x 2) 3)
        ((eq x 3) 4)
        ((eq x 4) 5)
        ((eq x 5) 6)
        ((eq x 6) 7)
        ((eq x 7) 8)
        ((eq x 8) 9)
        ((eq x 9) :f)
        (t (bottom '(not a digit:) x))))
```

(pred symbol) ⟶ symbol | :f

```
(define (pred x)
  (cond ((eq x 1) 0)
        ((eq x 2) 1)
        ((eq x 3) 2)
        ((eq x 4) 3)
        ((eq x 5) 4)
        ((eq x 6) 5)
        ((eq x 7) 6)
        ((eq x 8) 7)
        ((eq x 9) 8)
        ((eq x 0) :f)
        (t (bottom '(not a digit:) x))))
```

The **sum-of-digits** structure contains the sums of all combinations of digits in the form

`(sum . carry)`

The result of adding two digits *a* and *b* can be found in the *b*'th column of the *a*'th row of the structure. There are eleven columns in each row in order to support a single-bit carry value.

```
(define *sums-of-digits* '(
  ((0.0) (1.0) (2.0) (3.0) (4.0) (5.0) (6.0) (7.0) (8.0) (9.0) (0.1))
  ((1.0) (2.0) (3.0) (4.0) (5.0) (6.0) (7.0) (8.0) (9.0) (0.1) (1.1))
  ((2.0) (3.0) (4.0) (5.0) (6.0) (7.0) (8.0) (9.0) (0.1) (1.1) (2.1))
  ((3.0) (4.0) (5.0) (6.0) (7.0) (8.0) (9.0) (0.1) (1.1) (2.1) (3.1))
  ((4.0) (5.0) (6.0) (7.0) (8.0) (9.0) (0.1) (1.1) (2.1) (3.1) (4.1))
  ((5.0) (6.0) (7.0) (8.0) (9.0) (0.1) (1.1) (2.1) (3.1) (4.1) (5.1))
  ((6.0) (7.0) (8.0) (9.0) (0.1) (1.1) (2.1) (3.1) (4.1) (5.1) (6.1))
  ((7.0) (8.0) (9.0) (0.1) (1.1) (2.1) (3.1) (4.1) (5.1) (6.1) (7.1))
  ((8.0) (9.0) (0.1) (1.1) (2.1) (3.1) (4.1) (5.1) (6.1) (7.1) (8.1))
  ((9.0) (0.1) (1.1) (2.1) (3.1) (4.1) (5.1) (6.1) (7.1) (8.1) (9.1))
))
```

The **diffs-of-digits** structure contains the differences of all combinations of digits in the form

`(difference . borrow)`

The structure works in the same way as the above **sums-of-digits**. Because the "minus" operation is not commutative, it is important to look up the first operand to the difference operation in the rows and the second one in the columns of that row.

```
(define *diffs-of-digits* '(
  ((0.0) (9.1) (8.1) (7.1) (6.1) (5.1) (4.1) (3.1) (2.1) (1.1) (0.1))
  ((1.0) (0.0) (9.1) (8.1) (7.1) (6.1) (5.1) (4.1) (3.1) (2.1) (1.1))
  ((2.0) (1.0) (0.0) (9.1) (8.1) (7.1) (6.1) (5.1) (4.1) (3.1) (2.1))
  ((3.0) (2.0) (1.0) (0.0) (9.1) (8.1) (7.1) (6.1) (5.1) (4.1) (3.1))
  ((4.0) (3.0) (2.0) (1.0) (0.0) (9.1) (8.1) (7.1) (6.1) (5.1) (4.1))
  ((5.0) (4.0) (3.0) (2.0) (1.0) (0.0) (9.1) (8.1) (7.1) (6.1) (5.1))
  ((6.0) (5.0) (4.0) (3.0) (2.0) (1.0) (0.0) (9.1) (8.1) (7.1) (6.1))
  ((7.0) (6.0) (5.0) (4.0) (3.0) (2.0) (1.0) (0.0) (9.1) (8.1) (7.1))
  ((8.0) (7.0) (6.0) (5.0) (4.0) (3.0) (2.0) (1.0) (0.0) (9.1) (8.1))
  ((9.0) (8.0) (7.0) (6.0) (5.0) (4.0) (3.0) (2.0) (1.0) (0.0) (9.1))
))
```

The *%nth-item* function fetches the *d*'th item from the list *lst*. *D* must be a *digit* and not a zenlisp number. *Nth-item* is used to look up values in the above sum and difference tables.

(%nth-item digit list) ⟶ form

```
(define (%nth-item d lst)
  (cond ((eq d 0) (car lst))
        (t (%nth-item (pred d) (cdr lst)))))
```

%D+ adds two digits and a carry flag (represented by the digits `0` and `1`) and delivers a pair consisting of their sum and a new carry value. It basically implements a decimal single-digit full adder.

(%d+ digit$_1$ digit$_2$ 0|1) ⟶ '(*sum . carry*)

```
(define (%d+ a b carry)
  (let ((row (%nth-item b *sums-of-digits*)))
    (cond ((eq carry 1) (%nth-item a (cdr row)))
          (t (%nth-item a row)))))
```

%D- subtracts a digit *b* and a borrow flag from a digit *a*. The borrow flag is represented by the digits `0` and `1`. **%D-** and delivers a pair consisting of the difference and a new borrow flag.

(%d- digit$_1$ digit$_2$ 0|1) ⟶ '(*difference . borrow*)

```
(define (%d- a b carry)
  (let ((row (%nth-item a *diffs-of-digits*)))
    (cond ((eq carry 1) (%nth-item b (cdr row)))
          (t (%nth-item b row)))))
```

%D< is a predicate returning **:t** if the digit *a* has a smaller value than the digit *b*.

```
(define (%d< a b)
  (letrec
    ((dless
       (lambda (set)
         (cond ((null set)
                 (bottom '(not digits:) a b))
               ((eq a (car set))
                 (not (eq b (car set))))
               ((eq b (car set)) :f)
               (t (dless (cdr set)))))))
    (dless *digits*)))
```

Natural-p checks whether its argument is a natural number (a non-empty list of digits).

(natural-p form) ⟶ :t | :f

```
(define (natural-p x)
  (letrec
    ((lod-p
       (lambda (x)
         (cond ((null x) :t)
               ((atom x) :f)
               (t (and (digitp (car x))
                       (lod-p (cdr x))))))))
    (and (not (atom x))
         (lod-p x))))
```

N-natural converts a number to a natural number. Because there are only natural number at this point, this is an identity operation.

(n-natural natural) —> natural

```
(define n-natural id)
```

Normalize a natural number by removing leading zeroes.

(n-normalize natural) —> natural

```
(define (n-normalize x)
  (cond ((null (cdr x)) x)
        ((eq (car x) 0)
          (n-normalize (cdr x)))
        (t x)))
```

Check whether two natural numbers are in strict ascending order. **N<** uses an empty **let** in order to close over `%d<`. Thi construct is used in packages to protect internal symbols from accidental redefinition.

(n< natural$_1$ natural$_2$) —> :t | :f

```
(define n<
  (let ()
    (lambda (a b)
      (letrec
        ((d> (lambda (a b)
               (%d< b a)))
         (lt (lambda (a b r)
               (cond ((and (null a) (null b)) r)
                     ((null a) :t)
                     ((null b) :f)
                     (t (lt (cdr a)
                            (cdr b)
                            (cond ((%d< (car a) (car b)) :t)
                                  ((d> (car a) (car b)) :f)
                                  (t r)))))))))
        (lt (reverse a) (reverse b) :f)))))
```

The other ordering predicates can be derived from **n<** easily:

(n> natural$_1$ natural$_2$) —> :t | :f
(n<= natural$_1$ natural$_2$) —> :t | :f
(n>= natural$_1$ natural$_2$) —> :t | :f

```
(define (n> a b) (n< b a))
```

```
(define (n<= a b) (eq (n> a b) :f))

(define (n>= a b) (eq (n< a b) :f))
```

Check whether two natural numbers are equal.

(n= $natural_1$ $natural_2$) ⟶ :t | :f

```
(define (n= a b)
  (equal (n-normalize a)
         (n-normalize b)))
```

Add two natural numbers. The algorithm used here is basically the one that most people use when adding two numbers on a sheet of paper. It starts with the least significant digits and propagates a carry flag through the entire rows of digits. When one number has fewer digits than the other, the non-existent digits are substituted by 0.

(n+ $natural_1$ $natural_2$) ⟶ natural

```
(define n+
  (let ()
    (lambda (a b)
      (letrec
        ((add
           (lambda (a b c r)
             (cond ((null a)
                     (cond
                       ((null b)
                         (cond ((eq c 0) r)  ; no carry
                               (t (cons 1 r))))
                       (t (let ((sum (%d+ 0 (car b) c)))
                            (add ()
                                 (cdr b)
                                 (cdr sum)
                                 (cons (car sum) r))))))
                   ((null b)
                     (let ((sum (%d+ (car a) 0 c)))
                       (add (cdr a)
                            ()
                            (cdr sum)
                            (cons (car sum) r))))
                   (t (let ((sum (%d+ (car a) (car b) c)))
                        (add (cdr a)
                             (cdr b)
                             (cdr sum)
                             (cons (car sum) r))))))))
        (add (reverse a) (reverse b) 0 ())))))
```

Subtract two natural numbers. Similar to **n+** above.

(n- natural$_1$ natural$_2$) ⟶ natural

```
(define n-
  (let ()
    (lambda (a b)
      (letrec
        ((diff
           (lambda (a b c r)
             (cond ((null a)
                     (cond
                       ((null b)
                         (cond ((eq c 0) r)
                               (t (bottom '(negative difference)))))
                       (t (bottom '(negative difference)))))
                   ((null b)
                     (cond ((eq c 0)
                             (append (reverse a) r))
                           (t (diff a '(1) 0 r))))
                   (t (let ((delta (%d- (car a) (car b) c)))
                        (diff (cdr a)
                              (cdr b)
                              (cdr delta)
                              (cons (car delta) r))))))))
        (n-normalize (diff (reverse a) (reverse b) 0 ()))))))
```

Test a natural number for being zero or one. These functions are equivalent to **(=** *x* **0)** and **(=** *x* **1)** respectively, but much more efficient.

(n-zero natural) ⟶ :t | :f
(n-one natural) ⟶ :t | :f

```
(define (n-zero x)
  (and (eq (car x) 0)
       (null (cdr x))))

(define (n-one x)
  (and (eq (car x) 1)
       (null (cdr x))))
```

Multiply two natural numbers. The algorithm uses decimal left shift operations to multiple by 10.

(n* natural$_1$ natural$_2$) ⟶ natural

```
(define (n* a b)
  (letrec
    ((*10
       (lambda (x)
         (append x '#0)))
     (add-n-times
       (lambda (a b r)
         (cond ((n-zero (list b)) r)
               (t (add-n-times a (pred b) (n+ a r))))))
     (times
       (lambda (a b r)
         (cond ((null b) r)
               (t (times (*10 a)
                         (cdr b)
                         (add-n-times a (car b) r)))))))
    (cond ((n-zero a) '#0)
          (t (times a (reverse b) '#0)))))
```

Divide two natural numbers, giving a list of the form `(`*`quotient remainder`*`)`. The division algorithm works as follows:

– shift the divisor to the left until it has as many digits as the dividend;
– let **n** be the number of places by which the divisor was shifted;
– let the result **R** be **`'#0`**;
– do **n** times:
  – test how many times the divisor fits into the divident; name this number **q**;
  – subtract **q** times the divisor from the dividend;
  – append **q** to **R**; [23]
  – shift the divisor to the right by one digit.
– normalize the result **R**.

`(n-divide natural`$_1$ `natural`$_2$`)` $\longrightarrow$ **`'(`***`quotient remainder`***`)`**

```
(define (n-divide a b)
  (letrec
    ; Equalize the divisor B by shifting it to the left
    ; (multiplying it by 10) until it has the same number
    ; of digits as the dividend A.
    ; Return: (new divisor . base 1 shift count)
    ((eql
       (lambda (a b r s)
         (cond ((null a)
                 (cons (reverse r) s))
               ((null b)
                 (eql (cdr a)
                      ()
                      (cons 0 r)
                      (cons 'i s)))
               (t (eql (cdr a)
                       (cdr b)
                       (cons (car b) r)
                       s)))))
     ; Divide with quotient < 10
```

23 Yes, zenlisp numbers can be appended using **append** because they are ordinary lists.

```
      ; Return (A/B*B . A/B)
      (div10
        (lambda (a b r)
          (cond ((n< (car r) a)
                  (div10 a b (cons (n+ (car r) b)
                                   (n+ (cdr r) '#1))))
                ((equal (car r) a) r)
                (t (cons (n- (car r) b)
                         (n- (cdr r) '#1))))))
      ; X / 10
      (d10
        (lambda (x)
          (reverse (cdr (reverse x)))))
      (div
        (lambda (a b r)
          (cond ((null (cdr b))
                  (list (n-normalize r) a))
                (t (let ((quot (div10 a (car b) (cons '#0 '#0))))
                     (div (n- a (car quot))
                          (cons (d10 (car b)) (cddr b))
                          (append r (cdr quot)))))))))
    (cond ((n-zero b) (bottom 'divide-by-zero))
          ((n< a b) (list '#0 a))
          (t (div a (eql a b () '#i) '#0)))))
```

Divide two natural numbers and return only the quotient or the remainder.

(n-quotient natural$_1$ natural$_2$) ⟶ natural
(n-remainder natural$_1$ natural$_2$) ⟶ natural

```
(define (n-quotient a b) (car (n-divide a b)))

(define (n-remainder a b) (cadr (n-divide a b)))
```

Test a number for being even or odd. **Even** is basically **(n-zero (remainder *x* '#2))**, but more efficient.

(even natural) ⟶ natural
(odd natural) ⟶ natural

```
(define (even x)
  (and (memq (car (reverse x)) '#02468) :t))

(define (odd x) (eq (even x) :f))
```

Compute *x* raised to the *y*'th power.

(n-expt natural$_1$ natural$_2$) ⟶ natural

```
(define (n-expt x y)
  (letrec
    ((square
       (lambda (x)
         (n* x x)))
     (n-expt1
       (lambda (y)
         (cond ((n-zero y) '#1)
               ((even y)
                 (square (n-expt1 (n-quotient y '#2))))
               (t (n* x (square (n-expt1 (n-quotient y '#2)))))))))
    (n-expt1 (n-natural y))))
```

Compute the greatest natural number that is not greater than the square root of the given argument. This function uses Newton's method.

(n-sqrt natural) —> natural

```
(define (n-sqrt square)
  (letrec
    ((sqr
       (lambda (x last)
         (cond ((equal last x) x)
               ((equal last (n+ x '#1))
                 (cond ((n> (n* x x) square) (n- x '#1))
                       (t x)))
               (t (sqr (n-quotient (n+ x (n-quotient square x))
                                   '#2)
                       x))))))
    (sqr square '#0)))
```

Compute the length of a list. This function is in the **nmath** package and not in **base**, because it uses numbers, which are not a trivial concept as you can see in this file.

(length list) —> natural

```
(define (length x)
  (letrec
    ((len (lambda (x r)
            (cond ((null x) r)
                  (t (len (cdr x) (n+ r '#1)))))))
    (len x '#0)))
```

Compute the *greatest common divisor* and *least common multiple* of two natural numbers.

(n-gcd natural) —> natural
(n-lcm natural) —> natural

```
(define (n-gcd a b)
  (cond ((n-zero b) a)
        ((n-zero a) b)
        ((n< a b) (n-gcd a (n-remainder b a)))
        (t (n-gcd b (n-remainder a b)))))
```

```
(define (n-lcm a b)
  (let ((cd (n-gcd a b)))
    (n* cd (n* (n-quotient a cd)
               (n-quotient b cd)))))
```

Find the limit **k** of a list of numbers **L** so that **k** *op* **x** for each **x** that is a member of **L** (without **k** if *op* is imposes a strict order). *Op* must be a numeric predicate that imposes an order on **L**. When *op* = **<**, for example, *limit* returns the minimum of the list.

(limit function natural$_1$ natural$_2$ ...) ⟶ natural

```
(define (limit op a . b)
  (letrec
    ((lim (lambda (a)
            (cond ((null (cdr a)) (car a))
                  ((op (car a) (cadr a))
                    (lim (cons (car a) (cddr a))))
                  (t (lim (cdr a)))))))
    (lim (cons a b))))
```

Find the maximum and minimum of a list using the *limit* function.

(max list) ⟶ natural
(min list) ⟶ natural

```
(define (n-max . a) (apply limit n> a))

(define (n-min . a) (apply limit n< a))

(require 'iter)
```

The following definitions specify the preferred names of the natural number operations. For example, `*` should be used in user-level code instead of `n*`, because it is more readable, more flexible (because it is variadic), and because it does not depend on a specific math package.

The iterator functions from the **`iter`** package are used to make some of the binary **`nmath`** functions variadic. Only the `-` function is converted manually, because the natural “minus” operator does not make any sense with less than 2 arguments.

```
(define natural n-natural)

(define * (arithmetic-iterator n-natural n* '#1))

(define + (arithmetic-iterator n-natural n+ '#0))

(define (- . x)
  (cond ((or (null x) (null (cdr x)))
          (bottom '(too few arguments to n-natural -)))
        (t (fold (lambda (a b)
                   (n- (n-natural a) (n-natural b)))
```

```
                 (car x)
                 (cdr x)))))

(define < (predicate-iterator natural n<))

(define <= (predicate-iterator natural n<=))

(define = (predicate-iterator natural n=))

(define > (predicate-iterator natural n>))

(define >= (predicate-iterator natural n>=))

(define divide n-divide)

(define expt n-expt)

(define gcd (arithmetic-iterator natural n-gcd '#0))

(define lcm (arithmetic-iterator natural n-lcm '#1))

(define max n-max)

(define min n-min)

(define number-p natural-p)

(define one n-one)

(define quotient n-quotient)

(define remainder n-remainder)

(define sqrt n-sqrt)

(define zero n-zero)
```

13.4 integer math functions

The **imath** package extends the **nmath** package by adding support for negative numbers. Many of the functions defined here basically split integer numbers into signs and a natural numbers, rewrite the operation so that it can be carried out by the corresponding natural math function, compute the sign of the result and attach it to the intermediate result delivered by the **nmath** function.

The integer math package is contained in the file `imath.l`.

```
; zenlisp integer math functions
; By Nils M Holm, 2007, 2008
; Feel free to copy, share, and modify this code.
; See the file LICENSE for details.
; would use REQUIRE, but REQUIRE is in BASE
(cond ((defined 'base) :f)
      (t (load base)))

(define imath :t)

(require 'nmath)
```

Check whether a form represents an integer number. An integer number is either a natural number or a natural number prefixed with a plus (“+”) or minus (“−”) sign.

(integer-p form) ⟶ :t | :f

```
(define (integer-p a)
  (and (not (atom a))
       (or (natural-p a)
           (and (memq (car a) '#+-)
                (natural-p (cdr a))))))
```

Convert an integer or natural number to an integer.

(i-integer natural | integer) ⟶ integer

```
(define (i-integer a)
  (cond ((eq (car a) '+) (cdr a))
        ((eq (car a) '-) a)
        ((digitp (car a)) a)
        (t (bottom (list 'i-integer a)))))
```

Convert a positive integer or natural number to a natural number.

(i-natural natural | integer) ⟶ natural

```
(define (i-natural a)
  (cond ((eq (car a) '+) (cdr a))
        ((digitp (car a)) a)
        (t (bottom (list 'i-natural a)))))
```

Normalize an integer number by removing leading zeroes and a positive sign.

(i-normalize integer) ⟶ integer

```
(define (i-normalize x)
  (cond ((eq (car x) '+)
          (n-normalize (cdr x)))
        ((eq (car x) '-)
          (let ((d (n-normalize (cdr x))))
            (cond ((n-zero d) d)
                  (t (cons '- d)))))
        (t (n-normalize x))))
```

Check whether the given integer is negative. This function is equivalent to **(<** *x* **0)**, but more efficient.

(i-negative integer) —> :t | :f

```
(define (i-negative x) (eq (car x) '-))
```

Remove the sign of an integer, thereby returning its absolute value.

(i-abs integer) —> natural

```
(define (i-abs x)
  (cond ((i-negative x) (cdr x))
        ((eq (car x) '+) (cdr x))
        (t x)))
```

Check whether a given integer is equal to zero or one. Like their natural counterparts these functions are performance hacks.

(i-zero integer) —> :t | :f
(i-one integer) —> :t | :f

```
(define (i-zero x)
  (n-zero (i-abs x)))

(define (i-one x)
  (and (n-one (i-abs x))
       (neq (car x) '-)))
```

Negate an integer. This function is equivalent to **(-** **0** *x***)**, but more efficient.

(i-negate integer) —> integer

```
(define (i-negate x)
  (cond ((n-zero (i-abs x)) x)
        ((eq (car x) '-) (cdr x))
        ((eq (car x) '+) (cons '- (cdr x)))
        (t (cons '- x))))
```

Add two integer numbers. This function handles only the signs and uses **n+** and **n-** to compute actual sums. In order to be able to do all computations using natural numbers, it rewrites its operations as outlined in the following table:[24]

Original term	**Rewritten term**	
+a + +b	a + b	
+a + -b	a - |b|	if |a| > |b|
+a + -b	-(|b| - a)	if |a| <= |b|

24 |*x*| denotes the absolute value of *x*.

-a + +b	-(|a| - b)	if |a| > |b|
-a + +b	b - |a|	if |a| <= |b|
-a + -b	-(|a| + |b|)	

(i+ integer$_1$ integer$_2$) —> integer

```
(define (i+ a b)
  (cond ((and (not (i-negative a))
              (not (i-negative b)))
          (n+ (i-abs a) (i-abs b)))
        ((and (not (i-negative a))
              (i-negative b))
          (cond ((n> (i-abs a) (i-abs b))
                  (n- (natural a) (i-abs b)))
                (t (i-negate (n- (i-abs b) (natural a))))))
        ((and (i-negative a)
              (not (i-negative b)))
          (cond ((n> (i-abs a) (i-abs b))
                  (i-negate (n- (i-abs a) (natural b))))
                (t (n- (natural b) (i-abs a)))))
        (t (i-negate (n+ (i-abs a) (i-abs b))))))
```

Subtract two integer numbers. This function handles only the signs and delegates the computations to **n-** and **i+**. It rewrites its operations as follows:

Original term	**Rewritten term**	
+a - +b	-(|b| - |a|)	if |a| < |b|
+a - +b	|a| - |b|	if |a| >= |b|
+a - -b	a + |b|	
-a - +b	a + -b	
-a - -b	a + |b|	

(i- integer$_1$ integer$_2$) —> integer

```
(define (i- a b)
  (cond ((i-negative b)
          (i+ a (i-abs b)))
        ((i-negative a)
          (i+ a (i-negate b)))
        ((n< (i-abs a) (i-abs b))
          (i-negate (n- (i-abs b) (i-abs a))))
        (t (n- (i-abs a) (i-abs b)))))
```

Check whether two integer numbers are in strict ascending order. This function first checks the signs to compare the numbers and delegates its task to **n<** only if the signs are equal.

(i< integer$_1$ integer$_2$) ⟶ :t | :f

```
(define (i< a b)
  (cond ((i-negative a)
          (cond ((not (i-negative b)) :t)
                (t (n< (i-abs b) (i-abs a)))))
        ((i-negative b) :f)
        (t (n< (i-abs a) (i-abs b)))))
```

As usual, the remaining ordering predicates can be derived from the “less than” relation.

(i> integer$_1$ integer$_2$) ⟶ :t | :f
(i<= integer$_1$ integer$_2$) ⟶ :t | :f
(i>= integer$_1$ integer$_2$) ⟶ :t | :f

```
(define (i> a b) (i< b a))

(define (i<= a b) (eq (i> b a) :f))

(define (i>= a b) (eq (i< b a) :f))
```

Check whether two integers are equal.

(i= integer$_1$ integer$_2$) ⟶ :t | :f

```
(define (i= a b)
  (equal (i-normalize a)
         (i-normalize b)))
```

Multiply two integers. Handle only signs, delegate the rest to **n***.

(i* integer$_1$ integer$_2$) ⟶ integer

```
(define (i* a b)
  (cond ((zero a) '#0)
        ((eq (i-negative a) (i-negative b))
          (n* (i-abs a) (i-abs b)))
        (t (i-negate (n* (i-abs a) (i-abs b))))))
```

Divide two integers. Handle only signs, delegate the rest to **n-divide**. Like **n-divide**, this function returns both the quotient and the remainder in a list.

(i-divide integer$_1$ integer$_2$) ⟶ '(*quotient remainder***)**

```
(define (i-divide a b)
  (letrec
    ((sign
       (lambda (x)
         (cond ((eq (i-negative a) (i-negative b)) x)
               (t (cons '- x)))))
     (rsign
       (lambda (x)
         (cond ((i-negative a) (cons '- x))
               (t x))))
     (idiv
       (lambda (a b)
         (cond ((n-zero b) (bottom '(divide by zero)))
               ((n< (i-abs a) (i-abs b))
                 (list '#0 (rsign (i-abs a))))
               (t (let ((q (n-divide (i-abs a) (i-abs b))))
                    (list (sign (car q))
                          (rsign (cadr q))))))))))
    (idiv (i-integer a) (i-integer b))))
```

Divide two integers and return only the quotient or the remainder.

(i-quotient integer$_1$ integer$_2$) ⟶ integer
(i-remainder integer$_1$ integer$_2$) ⟶ integer

```
(define (i-quotient a b) (car (i-divide a b)))

(define (i-remainder a b) (cadr (i-divide a b)))
```

Compute the modulus of two integers. Because the modulus is specific to integers, there is no version with an “i” prefix.

(modulo integer$_1$ integer$_2$) ⟶ integer

```
(define (modulo a b)
  (let ((rem (i-remainder a b)))
    (cond ((i-zero rem) '#0)
          ((eq (i-negative a)
               (i-negative b))
            rem)
          (t (i+ b rem)))))
```

Compute *x* raised to the power of *y*. Handle only signs, delegate the rest to **n-expt**. I guess you got the idea by now...

(i-expt integer$_1$ integer$_2$) ⟶ integer

```
(define (i-expt x y)
  (letrec
    ((i-expt2
       (lambda (x y)
         (cond ((or (not (i-negative x))
                    (even y))
                 (n-expt (i-abs x) y))
               (t (i-negate (n-expt (i-abs x) y)))))))
    (i-expt2 (i-integer x) (natural y))))
```

Define integer maximum and minimum in terms of **limit**.

(i-max integer$_1$ integer$_2$ ...) ⟶ integer
(i-min integer$_1$ integer$_2$ ...) ⟶ integer

```
(define (i-max . a) (apply limit i> a))

(define (i-min . a) (apply limit i< a))
```

I-sqrt is similar to **n-sqrt**, but rejects negative operands.

(i-sqrt integer) ⟶ integer

```
(define (i-sqrt x)
  (cond ((i-negative x)
          (bottom (list 'i-sqrt x)))
        (t (n-sqrt x))))
```

The integer versions of **gcd** and **lcm** just cut off the signs.

(i-gcd integer ...) ⟶ integer
(i-lcm integer ...) ⟶ integer

```
(define (i-gcd a b)
  (n-gcd (i-abs a) (i-abs b)))

(define (i-lcm a b)
  (n-lcm (i-abs a) (i-abs b)))

(require 'iter)
```

As in the **nmath** package, the remainder of the file defines the preferred names of the math functions. Many functions defined in **nmath** get redefined here and some new ones are added.

The only notable modification is the extension of the "minus" operator. When the **-** function of **imath** is applied to a single argument, it negates that argument.

```
(define integer i-integer)

(define * (arithmetic-iterator integer i* '#1))

(define + (arithmetic-iterator integer i+ '#0))

(define (- . x)
  (cond ((null x)
          (bottom '(too few arguments to integer -)))
        ((eq (cdr x) ())
          (i-negate (car x)))
        (t (fold (lambda (a b)
                   (i- (integer a) (integer b)))
                 (car x)
                 (cdr x)))))
```

```
(define < (predicate-iterator integer i<))

(define <= (predicate-iterator integer i<=))

(define = (predicate-iterator integer i=))

(define > (predicate-iterator integer i>))

(define >= (predicate-iterator integer i>=))

(define abs i-abs)

(define divide i-divide)

(define expt i-expt)

(define gcd (arithmetic-iterator integer i-gcd '#0))

(define lcm (arithmetic-iterator integer i-lcm '#1))

(define max i-max)

(define min i-min)

(define natural i-natural)

(define negate i-negate)

(define negative i-negative)

(define number-p integer-p)

(define one i-one)

(define quotient i-quotient)

(define remainder i-remainder)

(define sqrt i-sqrt)

(define zero i-zero)
```

13.5 rational math functions

The **`rmath`** package extends the **`imath`** package by adding support for rational numbers. The functions defined here basically split rational numbers into numerators and denominators, rewrite the operations so that they can be done by the integer math functions and return a result that has the least applicable type.

This means that rational number operations may return a value that has a lesser type than ''rational'' if that type is sufficient to represent the result. For example:

```
(+ '#1/3 '#2/3) => '#1
```

Because the sum of 1/3 and 2/3 can be represented by a natural number, the rational **+** operator returns one.

The rational math package is contained in the file `rmath.l`.

```
; zenlisp rational math functions
; By Nils M Holm, 2007, 2008
; Feel free to copy, share, and modify this code.
; See the file LICENSE for details.

; would use REQUIRE, but REQUIRE is in BASE
(cond ((defined 'base) :f)
      (t (load base)))

(define rmath :t)

(require 'imath)
```

Extract the numerator and denominator of a rational number.

(numerator rational) ⟶ integer
(denominator rational) ⟶ integer

```
(define (numerator x)
  (reverse (cdr (memq '/ (reverse x)))))

(define (denominator x) (cdr (memq '/ x)))
```

Check whether a form represents a rational number.

(rational-p form) ⟶ :t | :f

```
(define (rational-p x)
  (and (listp x)
       (memq '/ x)
       (integer-p (numerator x))
       (integer-p (denominator x))))
```

Check whether a form represents a number (either rational or integer or natural).

(r-number-p form) ⟶ :t | :f

```
(define (r-number-p x)
  (or (integer-p x)
      (rational-p x)))
```

Create a rational number from a given numerator and denominator.

(make-rational integer$_1$ integer$_2$) ⟶ rational

```
(define (make-rational num den)
  (append num '#/ den))
```

Convert any type of number to a rational number. When the number is not already rational, add a denominator of `'#1`. Note again that numeric types are distinguished by *syntax*, so `'#5` is not a rational number, but `'#5/1` is one.

(rational number) ⟶ rational

```
(define (rational x)
  (cond ((rational-p x) x)
        (t (make-rational x '#1))))
```

Rational versions of the **zero** and **one** functions.

(r-zero number) ⟶ :t | :f
(r-one number) ⟶ :t | :f

```
(define (r-zero x)
  (cond ((rational-p x) (r= x '#0))
        (t (i-zero x))))

(define (r-one x)
  (cond ((rational-p x) (r= x '#1))
        (t (i-one x))))
```

Reduce a rational number to its least terms.

(%least-terms rational) ⟶ rational

```
(define (%least-terms x)
  (let ((cd (gcd (numerator x) (denominator x))))
    (cond ((r-one cd) x)
          (t (make-rational (quotient (numerator x) cd)
                            (quotient (denominator x) cd))))))
```

Convert rationals with denominators of `'#1` to a lesser type.

(%decay rational) ⟶ rational | integer

```
(define (%decay x)
  (cond ((r-one (denominator x))
          (numerator x))
        (t x)))
```

Normalize a rational number. This is explained in depth on pages 52f. The empty **let** closes over *%least-terms* and *%decay*.

(r-normalize number) —> rational | integer

```
(define r-normalize
  (let ()
    (lambda (x)
      (letrec
        ((norm-sign (lambda (x)
          (let ((num (numerator x))
                (den (denominator x)))
            (let ((pos (eq (i-negative num)
                           (i-negative den))))
              (make-rational (cond (pos (i-abs num))
                                   (t (cons '- (i-abs num))))
                      (i-abs den)))))))
        (cond ((rational-p x)
                (%decay (%least-terms (norm-sign x))))
              (t (i-normalize x)))))))
```

Convert a rational or integer number to type integer/natural. Unlike their integer counterparts (**`i-integer`** and **`i-natural`**), these versions also accept rationals that can be reduced to integers/naturals.

(r-integer number) —> integer
(r-natural number) —> natural

```
(define (r-integer x)
  (let ((xlt (+ '#0 x)))
    (cond ((rational-p xlt)
            (bottom (list 'r-integer x)))
          (t xlt))))

(define (r-natural x)
  (i-natural (r-integer x)))
```

Compute the absolute value of a rational number.

(r-abs number) —> rational | integer

```
(define (r-abs x)
  (cond ((rational-p x)
          (make-rational (i-abs (numerator x))
                         (i-abs (denominator x))))
        (t (i-abs x))))
```

Equalize the denominators of two rational numbers, so they can be added, subtracted or compared. Quick: what is greater, `'#123/456` or `'#213/789`? *Equalize* gives the answer:

```
(%equalize '#123/456 '#213/789) => '(#32349/119928 #32376/119928)
```

However, there is no need to use *%equalize* in user-level code, because it is used by the rational **<** predicate, which is introduced later in this section.

(%equalize rational$_1$ rational$_2$) ⟶ list

```
(define (%equalize a b)
  (let ((num-a (numerator a))
        (num-b (numerator b))
        (den-a (denominator a))
        (den-b (denominator b)))
    (let ((cd (gcd den-a den-b)))
      (cond
        ((r-one cd)
          (list (make-rational (i* num-a den-b)
                               (i* den-a den-b))
                (make-rational (i* num-b den-a)
                               (i* den-b den-a))))
        (t (list (make-rational (quotient (i* num-a den-b) cd)
                                (quotient (i* den-a den-b) cd))
                 (make-rational (quotient (i* num-b den-a) cd)
                                (quotient (i* den-b den-a) cd))))))))
```

You already know this principle from the **imath** package: **R+**, **r-**, and **r*** handle only things that are specific to operations on rational numbers and delegate the rest to their integer counterparts.

(r+ number$_1$ number$_2$) ⟶ rational | integer
(r- number$_1$ number$_2$) ⟶ rational | integer
(r* number$_1$ number$_2$) ⟶ rational | integer

```
(define r+
  (let ()
    (lambda (a b)
      (let ((factors (%equalize (rational a) (rational b)))
            (radd
              (lambda (a b)
                (r-normalize
                  (make-rational (i+ (numerator a) (numerator b))
                                 (denominator a))))))
        (radd (car factors) (cadr factors))))))

(define r-
  (let ()
    (lambda (a b)
      (let ((factors (%equalize (rational a) (rational b)))
            (rsub
              (lambda (a b)
                (r-normalize
                  (make-rational (i- (numerator a) (numerator b))
                                 (denominator a))))))
        (rsub (car factors) (cadr factors))))))

(define (r* a b)
  (let ((rmul
          (lambda (a b)
            (r-normalize
              (make-rational (i* (numerator a) (numerator b))
                             (i* (denominator a) (denominator b)))))))
    (rmul (rational a) (rational b))))
```

There is no integer version of **/**, but because

*(a/b) / (c/d) = (a*d) / (b*c)*

rational division is easily delegated to **i***.

(r/ number$_1$ number$_2$) ⟶ rational | integer

```
(define (r/ a b)
  (let ((rdiv
          (lambda (a b)
            (r-normalize
              (make-rational (i* (numerator a) (denominator b))
                             (i* (denominator a) (numerator b)))))))
    (cond ((r-zero b) (bottom (list 'r/ a b)))
          (t (rdiv (rational a) (rational b))))))
```

R< uses *%equalize* and **i<** to compare rational numbers. The other relational predicates are derived from it. You already know this procedure from the other math packages .

(r< number$_1$ number$_2$) ⟶ :t | :f
(r> number$_1$ number$_2$) ⟶ :t | :f
(r<= number$_1$ number$_2$) ⟶ :t | :f
(r>= number$_1$ number$_2$) ⟶ :t | :f

```
(define r<
  (let ()
    (lambda (a b)
      (let ((factors (%equalize (rational a) (rational b))))
        (i< (numerator (car factors))
            (numerator (cadr factors)))))))

(define (r> a b) (r< b a))

(define (r<= a b) (eq (r> a b) :f))

(define (r>= a b) (eq (r< a b) :f))
```

Check whether two numbers (no matter which type they have) are equal.

(r= number$_1$ number$_2$) ⟶ :t | :f

```
(define r=
  (let ()
    (lambda (a b)
      (cond ((or (rational-p a) (rational-p b))
              (equal (%least-terms (rational a))
                     (%least-terms (rational b))))
            (t (i= a b))))))
```

Raise *x* to the *y*'th power. Unlike its integer counterpart this function accepts negative exponents.

(r-expt number$_1$ number$_2$) ⟶ rational | integer

```
(define (r-expt x y)
  (letrec
    ((rx (cond ((i-negative (r-integer y))
                 (r/ '#1 (rational x)))
                (t (rational x))))
     (square
       (lambda (x)
         (r* x x)))
     (exp
       (lambda (x y)
         (cond ((r-zero y) '#1)
               ((even y)
                 (square (exp x (quotient y '#2))))
               (t (r* x (square (exp x (quotient y '#2)))))))))
    (exp rx (i-abs (r-integer y)))))
```

Check whether a number is negative.

(r-negative number) ⟶ :t | :f

```
(define (r-negative x)
  (cond ((rational-p x)
          (i-negative (numerator (r-normalize x))))
        (t (i-negative x))))
```

Negate a number.

(r-negate number) ⟶ rational | integer

```
(define (r-negate x)
  (cond ((rational-p x)
          (let ((nx (r-normalize x)))
            (make-rational (i-negate (numerator nx))
                           (denominator nx))))
        (t (i-negate x))))
```

Define the rational maximum and minimum functions in terms of **`limit`**.

(r-max number$_1$ number$_2$ ...) ⟶ rational | integer
(r-min number$_1$ number$_2$ ...) ⟶ rational | integer

```
(define (r-max . a) (apply limit r> a))

(define (r-min . a) (apply limit r< a))
```

Compute the square root of a number. The *precision* argument specifies the maximum error of the result. A value of 10, for example, means that the value returned by this function may not differ from the actual square root of the argument by a value that is greater than 10^{-10} (or 0.0000000001).

(r-sqrt number natural) ⟶ rational | integer

```
(define (r-sqrt square precision)
  (let ((e (make-rational '#1 (r-expt '#10 (r-natural precision)))))
    (letrec
      ((sqr (lambda (x)
              (cond ((r< (r-abs (r- (r* x x) square))
                         e)
                      x)
                    (t (sqr (r/ (r+ x (r/ square x))
                                '#2)))))))
      (sqr (n-sqrt (r-natural square))))))

(require 'iter)
```

As in the other math packages, the remainder of the file defines the preferred names of the functions. The **-** and **/** functions accept at least one argument. When **/** is applied to a single argument, it returns its reciprocal value (1/x).

```
(define * (arithmetic-iterator rational r* '#1))

(define + (arithmetic-iterator rational r+ '#0))

(define (- . x)
  (cond ((null x)
          (bottom '(too few arguments to rational -)))
        ((eq (cdr x) ()) (r-negate (car x)))
        (t (fold (lambda (a b)
                   (r- (rational a) (rational b)))
                 (car x)
                 (cdr x)))))

(define (/ . x)
  (cond ((null x)
          (bottom '(too few arguments to rational /)))
        ((eq (cdr x) ())
          (/ '#1 (car x)))
        (t (fold (lambda (a b)
                   (r/ (rational a) (rational b)))
                 (car x)
                 (cdr x)))))

(define < (predicate-iterator rational r<))

(define <= (predicate-iterator rational r<=))
```

```
(define = (predicate-iterator rational r=))

(define > (predicate-iterator rational r>))

(define >= (predicate-iterator rational r>=))

(define abs r-abs)

(define *epsilon* '#10)

(define expt r-expt)

(define integer r-integer)

(define max r-max)

(define min r-min)

(define natural r-natural)

(define negate r-negate)

(define negative r-negative)

(define number-p r-number-p)

(define one r-one)

(define (sqrt x) (r-sqrt x *epsilon*))

(define zero r-zero)
```

A.1 tail call rules

A *tail-recursive* program is a program that recurses by using tail calls exclusively.

A tail call is a function application that is in a tail position.

This is an exhaustive list of tail positions in zenlisp:

(lambda (...) (function ...))
The outermost function application in a function body.

(let (...) body)
The bodies of **let**.

(letrec (...) body)
The bodies of **letrec**.

(apply function ...)
Applications of **apply**.

(and ... expression)
The *last* argument of **and**.

(or ... expression)
The *last* argument of **or**.

(cond ... (predicate body) ...)
Each body of **cond**.

Note that tail call rules may be combined. The application of **f** in the following example is in a tail position:

```
(lambda ()
  (let ()
    (cond (t (or :f :f (f))))))
```

A.2 zenlisp functions

The following symbols are used in this summary:

Symbol	**Meaning**
`alist`	an association list
`expr`	any type of expression
`form`	any type of form (unevaluated)
`fun`	a function or closure
`name`	a symbol (unevaluated)
`pair`	a pair
`symbol`	a symbol
`a | b`	either *a* or *b*
`a...`	zero, one, or multiple instances of *a*

A.2.1 definitions

`(define name expr)`
Define constant *name* with value *expr*.

(define ($name_1$ $name_2$ ...) expr)
Define function $name_1$ with optional variables $name_2$... and body *expr*.

`(defined symbol)`
Test whether *symbol* is defined.

`(lambda (name ...) expr)   |   (lambda name expr)`
Create closure with variables *name...* or *name* and body *expr*.

(let (($name_1$ $expr_1$) ...) expr)
Create an environment with the bindings $name_i$=$expr_i$... and evaluate *expr* in that environment.

(letrec (($name_1$ $expr_1$) ...) expr)
Create an environment with the recursive bindings $name_i$=$expr_i$... and evaluate *expr* in that environment.

`(quote form)`
Create a datum.

`(recursive-bind alist)`
Fix recursive bindings in environments.

A.2.2 control

`(and expr ...)`
Reduce expressions. Return the first one giving **`:f`** or the last one.

`(apply fun expr ... list)`
Apply *fun* to the optional expressions and the members of *list*.

`(bottom expr ...)`
Reduce to an undefined value.

(cond (expr$_p$ expr) ...)
Reduce to the first *expr* whose associated *expr$_p$* evaluates to truth.

`(eval expr)`
Reduce *expr* to its normal form.

`(or expr ...)`
Reduce expressions. Return the first one evaluating to truth or the last one.

A.2.3 lists

`(append list ...)`
Append lists.

`(assoc expr alist)`
Find association with key=*expr* in association list *alist*; else return **`:f`**.

`(assq symbol alist)`
Find association with key=*symbol* in association list *alist*; else return **`:f`**.

`(caar pair)` ... **`(cddddr pair)`**
Extract parts of nested pairs. **`Caar`** = car of car, **`cadr`** = car of cdr, etc.

`(car pair)`
Extract car part of *pair*.

`(cdr pair)`
Extract cdr part of *pair*.

(cons expr$_1$ expr$_2$)
Construct fresh pair **'(***expr$_1$* . *expr$_2$***)**.

(equal expr$_1$ expr$_2$)
Test whether *expr$_1$* is equal to (looks the same as) *expr$_2$*.

(explode symbol)
Decompose a symbol into a list of single-character symbols.

(fold fun expr list)
Fold *fun* over *list* with base value *expr*. Left-associative version.

(fold-r fun expr list)
Fold *fun* over *list* with base value *expr*. Right-associative version.

(implode list)
Compose a symbol from a list of single-character symbols.

(list expr ...)
Create a list with the given members.

(listp expr)
Test whether *expr* is a proper (non-dotted) list.

(map fun list$_1$ list$_2$ ...)
Map function *fun* over the given lists.

(member expr list)
Find the first sublist of *list* starting with *expr*, else return `:f`.

(memq symbol list)
Find the first sublist of *list* starting with *symbol*, else return `:f`.

(null expr)
Test whether *expr* is `()`.

(reverse list)
Return a reverse copy of *list*.

A.2.4 miscellanea

(atom expr)
Test whether *expr* is atomic (either a symbol or `()`).

(eq expr$_1$ expr$_2$)
Test whether *expr$_1$* and *expr$_2$* are identical.

(id expr)
Identity function (return *expr*).

(neq expr$_1$ expr$_2$)
Test whether *expr$_1$* and *expr$_2$* are *not* identical.

(not expr)
Test whether *expr* is identical to **:f** (logical “not”).

A.2.5 packages

(require symbol)
Load the given package if not already in memory.

A.2.6 meta functions

******
(OK, this it not really a function.) This variable is always bound to the latest toplevel result, i.e. the normal form that was most recently printed by the interpreter.

(closure-form args | body | env)
Control how much of a closure is printed.

(dump-image name)
Dump workspace image to file *name*. Reload the image by passing *name* to zenlisp.

(gc)
Run garbage collection and return statistics.

(load name)
Load definitions from the file *name*.

(quit)
End a zenlisp session.

(stats expr)
Reduce *expr* to normal form and return some statistics.

(symbols)
Return a list of all symbols in the symbol table.

(trace name) | (trace)
Trace the function with the given *name*. **(Trace)** switches tracing off.

(verify-arrows :t | :f)
Turn verification of **=>** operators on or off.

A.3 math functions

Symbols used in this summary:

Symbol	**Meaning**
`x`	any number
`r`	rational number
`i`	integer number
`n`	natural number
`[x]`	*x* is optional
`a|b`	either *a* or *b*
`a...`	zero, one, or multiple instances of *a*

Function	**Returns...**
`(* x ...) => x`	product
`*epsilon* => n`	$\log_{10}$ of precision of **`sqrt`**
`(+ x ...) => x`	sum
`(- x1 x2 x3 ...) => x`	difference
`(- x) => x`	negative number
`(/ x1 x2 x3 ...) => x`	ratio
`(< x1 x2 x3 ...) => :t|:f`	**`:t`** for strict ascending order
`(<= x1 x2 x3 ...) => :t|:f`	**`:t`** for strict non-descending order
`(= x1 x2 x3 ...) => :t|:f`	**`:t`** for equivalence
`(> x1 x2 x3 ...) => :t|:f`	**`:t`** for strict descending order
`(>= x1 x2 x3 ...) => :t|:f`	**`:t`** for strict non-ascending order
`(abs x) => x`	absolute value
`(denominator r) => i`	denominator
`(divide i1 i2) => '(i3 i4)`	quotient *i3* and remainder *i4*
`(even i) => :t|:f`	**`:t`**, if *i* is even
`(expt x i) => x`	*x* to the power of *i*
`(gcd i1 i2 ...) => n`	greatest common divisor
`(integer x) => i`	an integer with the value *x*
`(integer-p x) => :t|:f`	**`:t`**, if *x* is integer
`(lcm i1 i2 ...) => n`	least common multiple
`(length list) => n`	length of a list
`(max x1 x2 ...) => x`	maximum value
`(min x1 x2 ...) => x`	minimum value
`(modulo i1 i2) => i3`	modulus

`(natural x) => n`	a natural with the value *x*
`(natural-p x) => :t|:f`	**`:t`**, if *x* is natural
`(negate i|r) => i|r`	negative value
`(negative x) => :t|:f`	**`:t`**, if *x* is negative
`(number-p expr) => :t|:f`	**`:t`**, if *expr* represents a number
`(numerator r) => i`	numerator
`(odd i) => :t|:f`	**`:t`**, if *i* is not even
`(one x) => :t|:f`	**`:t`**, if *x* equals one
`(quotient i1 i2) => i`	quotient
`(rational x) => r`	a rational with the value *x*
`(rational-p x) => :t|:f`	**`:t`**, if *x* is rational
`(remainder i1 i2) => i`	division remainder
`(sqrt n) => x`	square root, see also **`*epsilon*`**
`(zero x) => :t|:f`	**`:t`**, if *x* equals zero

A.4 working with zenlisp

Meta functions alter the state of the zenlisp system.

The **`load`** meta function reads a text file and reduces all expressions contained in that file to their normal forms.

Given a file named `palindrome.l` containing the lines

```
(define (palindrome x)
  (append x (reverse x)))
```

the function application

```
(load palindrome)
```

will load the above definition.

`Load` automatically appends the `.l` suffix to the file name.

Loading the same file again will update all definitions of that file.

When the file name begins with a tilde, **`load`** loads a zenlisp package from a pre-defined location. For instance

```
(load ~nmath)
```

loads the natural math functions into the zenlisp system.

The actual location of the system packages is specified by the `$ZENSRC` environment variable.

While **`load`** is typically used to load packages interactively, **`require`** is used make a program dependent on a *package* [page 62].

A program beginning with the function application

```
(require '~rmath)
```

depends on the **`rmath`** package.

Because **`require`** is a lambda function (and not an internal pseudo function), its argument has to be quoted. Unlike **`load`**, **`require`** never loads a package twice:

```
(require '~rmath) => :t
(require '~rmath) => :f
```

Hence it can be used to load mutually dependent packages.

The **`dump-image`** meta function dumps the complete zenlisp workspace to a file:

```
(load ~rmath)
(load ~amk)
(dump-image my-workspace)
```

This session creates a new image file named `my-workspace` which contains the **`rmath`** and **`amk`** packages.

To load an image, pass the image file name to zenlisp (`%` is the prompt of the Unix shell):

```
% zl my-workspace
```

The **`trace`** meta function makes the interpreter trace applications of a specific function:

```
(define (d x) (or (atom x) (d (cdr x))))
(d '#xyz) => :t
(trace d) => :t
(d '#xyz)
+ (d #xyz)
+ (d #yz)
+ (d #z)
+ (d ())
=> :t
```

Applying **`trace`** to no arguments switches tracing off:

```
(trace) => :t
```

The **`stats`** meta function measures the resources used during the reduction of an expression:

```
(stats (append '#abc '#def))
=> '(#abcdef #240 #1,213 #0)
```

Its normal form is a list containing the following information:

'(*normal-form steps nodes gcs*)

where

– *normal-form* is the normal form of the expression to evaluate;
– *steps* is the number of reduction steps performed;
– *nodes* is the number of nodes allocated during reduction;
– *gcs* is the number of garbage collections performed during reduction.

The **verify-arrows** function switches verification mode on or off. Passing **:t** to it enables verification, **:f** disables it:

```
(verify-arrows t) => :t
```

In verification mode, **=>** operators are verified by making sure that the normal form on the lefthand side of each **=>** matches its righthand side:

```
(cons 'heads 'tails) => '(heads . tails) ; OK
(cons 'heads 'tails) => 'foo             ; FAIL
```

As long as an expression reduces to the expected normal form, nothing special happens. When the forms do not match, though, an error is reported by zenlisp:

```
(cons 'heads 'tails) => 'foo
=> '(heads . tails)
* 2: REPL: Verification failed; expected: 'foo
```

In non-verifying mode **=>** introduces a comment to the end of the line (like **;**), thereby facilitating the cutting and pasting of expressions.

The **quit** meta function ends a zenlisp session:

```
(quit)
```

For obvious reasons, **(quit)** has no normal form.

A.4.1 the development cycle

Although you can enter whole programs at the read-eval-print loop (REPL), doing so might turn out to be a bit inconvenient, because the zenlisp REPL lacks all but the most rudimentary editing features.

So it is recommended that you use a text editor of your choice to enter or modify zenlisp programs. For instance, you may have typed the following code and saved it to the file `hanoi.l`:[25]

```
(require '~nmath)
```

25 Zenlisp source code *must* have a `.l` suffix or the interpreter cannot load it.

```
(define (hanoi n)
  (letrec
    ((h (lambda (n from to via)
          (cond ((zero n) ())
                (t (append (h (- n '#1) from via to)
                           (list (liat from to))
                           (h (- n '#1) via to from))))))))
    (h n 'from 'to 'via)))
```

The most practical approach to test the program is to keep a window or virtual terminal open that runs a zenlisp process. After saving the above program, go to the zenlisp terminal and load the program (user input is in italics):

```
(load hanoi)
* hanoi.l: 10: REPL: wrong argument count: (define (hanoi n) (letrec ((h
(lambda (n from to via) (cond ((zero n) ()) (t (append (h (- n '#1) from
via to) (list (liat from to)) (h (- n '#1) via to from)))))))) (h n 'from
'to 'via))
```

The interpreter detects some syntax errors already at load time, like the above one. It informs you that line 10 of the file `hanoi.l` contained a function application with a wrong number of arguments. The error occurred at the top level (REPL).

At this point, an editor that can highlight or otherwise match parentheses is a great help. Using it, you will quickly discover that the **`letrec`** form of the file ends prematurely in the following line:

```
                           (h (- n '#1) via to from))))))))
```

Deleting the superflous closing parenthesis should fix the problem and indeed, when you reload the program in the interpreter window, it works fine (so far):

```
(load hanoi)
=> :t
```

The next step is to feed some input to the program:

```
(hanoi '#3)
* 4: h: symbol not bound: liat
* Trace: h h
```

Each line printed by the interpreter that begins with an asterisk indicates trouble. In the above case it is caused by an unbound symbol named *liat*. The line number is useless in this case, because it refers to the REPL and not to a file. However, we can see that the error occurred in the *h* function. Indeed this function contains a misspelled instance of **`list`**, so this error is a simple typo. Just return to the editor session and fix this bug, too:

```
                           (list (list from to))
```

Then go back to the interpreter and load the code again. This time, it should work fine:

```
(load hanoi)
=> :t
(hanoi '#3)
=> '((from to) (from via) (to via) (from to) (via from) (via to) (from to))
```

Great. BTW: this program is the native zenlisp version of the Towers of Hanoi MEXPR program shown on page 162. Now let us try something more ambitious:

```
(hanoi '#20)
```

Maybe, this was a bit *too* ambitous, but pressing `Control` and `c` will stop the program:

```
^C
* hanoi.l: 6: append2: interrupted
* Trace: fold2 h h h h h h h h h
```

And once again with a smaller value:

```
(hanoi '#14)
=> ...lots of output...
```

In case you just wanted to know how many moves it takes, there is no need to rerun the program. Zenlisp always binds the latest result to the variable `**` after printing it on the REPL, so you can just type:

```
(length **)
=> '#16383
```

And if you want to know why the *hanoi* function is so slow, try this:

```
(gc)
=> '(#76285 #63103)
```

The first number is the number of unused ‘‘nodes’’ and the second one is the number of used ‘‘nodes’’. A *node* is an abstract unit which zenlisp uses to allocate memory. When the number of used nodes is larger than the number of free nodes (or close to that number), the interpreter will start to spend significant time trying to reclaim unused memory. Running the interpreter with a larger memory pool will give the program a performance boost:

```
(quit)
% zenlisp -b 1024K
zenlisp 2008-02-02 by Nils M Holm
(load hanoi)
(hanoi '#14)
=> ...lots of output...
(gc)
=> '(#993789 #63098)
```

A.5 zenlisp for the experienced schemer

Zenlisp is very much like a small subset of the *Scheme* programming language as defined in the Revised^5 Report on the Algorithmic Language Scheme (R5RS), but there are some more or less subtle differences. This is a summary of the most important differences.

The only types are the pair and the atom, atoms are symbols or **`()`**.

The canonical truth value is **`:t`** and falsity is **`:f`**.

Bodies are single expressions, there is no `begin`.

`Cond` must have a default clause.

`()` does not have to be quoted (but doing so does not hurt).

Predicates do not have a trailing "?", so you write **`(zero x)`** instead of **`(zero? x)`**.

Lists of single-character symbols may be "condensed": **`'(x y z)`** = **`'#xyz`**

Numbers are lists: **`(+ '#12 '#34) => '#46`**

Special form handlers are first-class objects: **`lambda => {internal lambda}`**

`Apply` works fine with special forms: **`(apply or '(:f :f :f 'foo)) => 'foo`**

`Letrec` is defined in terms of **`let`** and **`recursive-bind`** instead of **`let`** and `set!`.

Closures have first class environments:
`(cadddr (lambda () x)) => '((x . {void}))`

All data is immutable, there is no `set!`.

A.6 answers to some questions

Q1, Page 63

Pro: **`(headp ()`** *x***`)`** should yield **`:t`** for any *x*, because all lists have zero leading elements in common.

Contra: **`(headp ()`** *x***`)`** should yield **`:f`** for any *x*, because **`(cons ()`** *x***`)`** $\neq$ *x*.

The confusion arises because the term "head of a list" is used to name two different things: the car part of a pair and the leading elements of a list.

When *headp* was named something like *common-leading-members-p*, things would become clearer. **`(Common-leading-members () x)`** should always yield **`:t`**.

The bonus question remains: find a better name for *headp* that is shorter than *common-leading-members-p*.

Q2, Page 64

This version of *count* would be confusing, because it would count trailing **`()`**s of proper lists, so

```
(count '()) => '#1  ; fine
```

but

```
(count '(a b c))   => '#4  ; oops
(count '(a (b) c)) => '#5  ; oops
```

Q3, Page 66

Yes, *flatten* can be transformed to a function using tail calls exclusively. Any function can be transformed in such a way. In the case of *flatten*, this transformation would not improve efficiency, though, because the function constructs a tree structure and so it has to use some means of structural recursion.

Q4, Page 68

In this particular case, the use of **`append`** is not critical, because the first argument of **`append`** does not grow:

```
(trace append) => :t
(fold-right (lambda (x y z) (list 'op x y z))
            '0
            '(a b c)
            '(d e f))
+ (append #cf #0)
+ (append #be ((op c f 0)))
+ (append #ad ((op b e (op c f 0))))
=> '(op a d (op b e (op c f 0)))
```

Q5, Page 68

Substitution is not a proper substitute for beta reduction, because it would replace both free and bound variables:

```
(substitute '(list (lambda (x) x)) '((x . #17)))
=> '(list #17 (lambda (#17) #17))
```

Q6, Page 70

Insertion sort needs about $n^2/2$ steps when sorting an already sorted list of n elements:

```
(load ~nmath)
(load ~isort)
(trace sort)
(isort < '(#1 #2 #3 #4 #5))
+ (sort (#1 #2 #3 #4 #5) ())
+ (sort (#2 #3 #4 #5) (#1))  ; element inserted after 1 step
+ (sort (#3 #4 #5) (#1 #2))  ; element inserted after 2 steps
+ (sort (#4 #5) (#1 #2 #3))  ; element inserted after 3 steps
+ (sort (#5) (#1 #2 #3 #4))  ; element inserted after 4 steps
+ (sort () (#1 #2 #3 #4 #5)) ; element inserted after 5 steps
=> '(#1 #2 #3 #4 #5)         ; total = 15 steps
```

It needs *n* steps to sort a reverse sorted list:

```
(isort < '(#5 #4 #3 #2 #1))
+ (sort (#5 #4 #3 #2 #1) ())
+ (sort (#4 #3 #2 #1) (#5))  ; element inserted after 1 step
+ (sort (#3 #2 #1) (#4 #5))  ; element inserted after 1 step
+ (sort (#2 #1) (#3 #4 #5))  ; element inserted after 1 step
+ (sort (#1) (#2 #3 #4 #5))  ; element inserted after 1 step
+ (sort () (#1 #2 #3 #4 #5)) ; element inserted after 1 step
=> '(#1 #2 #3 #4 #5)         ; total = 5 steps
```

The average number of steps required by *isort* is the average of these extreme values.

Because the run time of *isort* is not easily predictable within the range limited by these extremes, it is not practicable as a sorting algorithm.

Q7, Page 73

Strictly speaking, most sorting functions use non-strict predicates to sort sets, even if strict predicates are more popular in the real world. For instance,

```
(S < '(#1 #1)) => bottom
```

should hold for any sorting algorithm *S*, because no strict order can be imposed on a set containing equal members.

So if you want a mathematically correct notation, non-strict predicates are the way to go. If you want to swim with the stream, strict predicates are what you want.

Q8, Page 73

This version of *insert* is stable when using non-strict predicates:

```
(define (insert p x a)
  (letrec
    ((ins
       (lambda (a r)
         (cond ((null a)
                 (reverse (cons x r)))
               ((not (p x (car a)))
                 (ins (cdr a) (cons (car a) r)))
               (t (append (reverse (cons x r)) a))))))
    (ins a ())))
```

Any sorting algorithm can be converted from stability under strict to stability under non-strict predicates and vice versa by using the following scheme.

In every sorting algorithm there is a point where elements are compared:

```
(cond ((p x y) sort-them)
      (t       already-sorted))
```

This part of code is modified by negating the predicate and swapping the branches:

```
(cond ((not (p x y)) already-sorted)
      (t             sort-them))
```

Q9, Page 77

Because *unsort* picks members from pretty random positions of the source list, it has the same average complexity as *isort*, which inserts members at pretty random positions.

Q10, Page 80

Omitting the clause would turn *for-all* into a predicate.

Q11, Page 83

The complexity of *combine* depends entirely on its second argument: the size of the source set. Iterating the first argument yields a degenerate curve:

```
(map length
     (map (lambda (x) (combine x '#abcde))
          '(#1 #2 #3 #4 #5 #6 #7 #8 #9 #10)))
=> '(#5 #10 #10 #5 #1 #0 #0 #0 #0 #0)
```

The *tails-of* function is used to iterate over sets of different sizes in order to estimate the complexity of *combine*:

```
(map length
     (map (lambda (x) (combine '#5 x))
          (reverse (tails-of '#0123456789abcdef))))
=> '(#0 #0 #0 #0 #1 #6 #21 #56 #126 #252 #462 #792 #1287 #2002 #3003)
```

Because the distance between the points keeps increasing on the y-axis of the curve, the complexity is worse than linear. It is probably even worse than $O(n^2)$, because $16^2 = 256$ and the above function yields 3003 for a set size of 16. The complexity is probably better than $O(2^n)$, though, because $3003<2^{16}$ and the average increase of the value is less than two times the previous value.

To find out whether *combine* exhibits polynomial behavior with a large exponent or exponential behavior with a small exponent, more precise analysis are required.

Because *combine* uses structural recursion (recursion through **`map`** is a dead giveaway for structural recursion), exponential complexity seems highly probable, though.

A similar apporach can be used to estimate the complexity of *combine**.

Q12, Page 85

Here is the modified *permutations* function with the additional clause rendered in bodface characters:

```
(define (permutations set)
  (cond
    ((null set) ())
    ((null (cdr set)) (list set))
    ((null (cddr set)) (rotations set))
    (t (apply append
                (map (lambda (rotn)
                       (map (lambda (x)
                              (cons (car rotn) x))
                            (permutations (cdr rotn))))
                     (rotations set))))))
```

Yes, the modification makes sense, because it reduces the average run time of *permutations* to about 67% of the original version. [26]

No matter how clever this optimization is, the complexity of *permutations* remains unchanged, because *n!* permutations still have to be created for a set of *n* elements:

```
(map length
     (map permutations
          (reverse (tails-of '#abcdefg))))
=> '(#1 #2 #6 #24 #120 #720 #5040)
```

Q13, Page 87

A trivial yet efficient approach to computing *n!* in zenlisp is:

```
(apply * (iota '#1 n))
```

26 See chapter 2.7 of "Sketchy LISP" (Nils M Holm; Lulu Press, 2008) for a detailed discussion.

It uses the *iota* function from page 86. Can you explain why it is even more efficient than the recursive product method, at least in zenlisp?

Q14, Page 90

Indeed, why not:

```
(filter (lambda (x)
          (or (null (cdr x))
              (apply >= x)))
        (part '#4))
=> '((#1 #1 #1 #1) (#2 #1 #1) (#2 #2) (#3 #1) (#4))
```

Q15, Page 93

$$
\begin{aligned}
3^{(6)}3 &= 3^{(5)}3^{(5)}3\\
&= 3^{(5)}3^{(4)}3^{(4)}3\\
&= 3^{(5)}3^{(4)}3\text{^}3\text{^}3\\
&= 3^{(5)}3^{(4)}3\text{^}27\\
&= 3^{(5)}3^{(4)}7625597484987
\end{aligned}
$$

The righthand part of the last line in the above equation is once again a power tower of the height 7,625,597,484,987 (as shown on page 92). But this time, that vast number merely describes the number of hyper-5 operations to apply to the factor of 3:

```
3^(5) ..................................... ^(5)3
|                                              |
+---------- 3^(4)7625597484987 times ----------+
```

The number $3^{(6)}3$ is indeed very hard to describe, even in terms of a power-power tower.

Q16, Page 95

Here is a generator that produces the tails of a list:

```
((generator '#abcdef cdr)) => '(#abcd . {closure ()})
                 (next **) => '(#bcd . {closure ()})
                 (next **) => '(#cd . {closure ()})
                 (next **) => '(#d . {closure ()})
                 (next **) => bottom
```

When the end of the list is reached, the generator yields bottom, because **`cdr`** cannot take the cdr part of **`()`**. See the concept of “streams” that is introduced in the following section for a more elegant solution.

Q17, Page 99

Using **`fold-r`**, the *append-streams** function is easily implemented:

```
(define (append-streams* . a)
  (fold-r append-streams :f a))
```

Applying the function to zero arguments yields the "end of stream" indicator **`:f`**, just like **`(append)`** yields the "end of list" indicator **`()`**.

Q18, Page 99

All non-recursive stream functions (plus *stream* itself) can be applied to infinite streams safely. These functions are

```
stream  map-stream  filter-stream  append-streams
```

Of course appending a finite stream to an infinite stream renders the finite stream inaccesible.

As outlined in the text, *stream->list* is not safe. The *stream-member* function is only safe if a member with the desired property is guaranteed to exist in the stream. The following application of *stream-member* reduces to bottom:

```
(stream-member even
               (stream '#1 id all (lambda (x) (+ '#2 x)) none :f)
               :f)
```

Q19, Page 105

The signature of the signature of a record has only members of the type *symbol*:

```
(record-signature (record '(food apple) '(weight #550) '(vegetarian :t)))
=> '((%record) (food symbol) (weight number) (vegetarian boolean))

(record-signature **)
=> '((%record) (food symbol) (weight symbol) (vegetarian symbol))
```

Because `symbol` is a symbol, this result is a fixed point of the *record-signature* function: passing it to the function will yield the same result over and over again.

Q20, Page 117

The tree structured is formed by the call tree of the program. The inner nodes of the trees are the functions of the parser.

Q21, Page 117

Here is a grammar that gives the unary minus a lower precedence than the power operator, so that

$-x^2 = -(x^2)$

Differences to the grammar on page 110 are rendered in boldface characters.

```
<sum> := <term>
       |  <term> '+' <sum>
       |  <term> '-' <sum>

<term> := <negation>
        |  <negation> '*' <term>
        |  <negation> <term>
        |  <negation> '/' <term>

<negation> := <power>
            |  '-' <power>

<power> := <factor>
         |  <factor> '^' <power>

<factor> := symbol
          |  number
          ;  removed the '-' <factor> rule
          |  '[' <sum> ']'
```

Q22, Page 121

Here is a version of the *infix* subfunction of *prefix->infix* that places parentheses around all binary operators. The modified *prefix->infix* function does not make use of the *paren* and *add-parens* subfunctions.

Differences to the original version print in boldface characters:

```
(infix
  (lambda (x)
    (cond
      ((numeric-p x)
        (cadr x))
      ((symbol-p x)
        (list x))
      ((and (eq (car x) '-)
            (not (atom (cdr x)))
            (null (cddr x)))
        (append '#- (infix (cadr x))))
      ((and (eq (car x) '[])
            (not (atom (cdr x)))
```

```
                 (null (cddr x)))
             (append '#[ (infix (cadr x)) '#]))
           ((and (not (atom x))
                 (not (atom (cdr x)))
                 (not (atom (cddr x)))
                 (null (cdddr x))
                 (function-p (car x)))
             (append '#[
                     (infix (cadr x))
                     (list (cdr (assq (car x) ops)))
                     (infix (caddr x))
                     '#]))
           (t (bottom (list 'syntax 'error: x))))))
```

Adding parentheses to all operations explicitly is useful when translating source code of one language to source code of another language that has different precedence rules.

Q23, Page 121

RPN (reverse polish notation, postfix notation) does not need parentheses, because precedence is expressed by the ordering of operators and operands:

Infix	**RPN**	zenlisp
`((a + b) * c) - d`	`a b + c * d -`	`(- (* (+ a b) c) d)`
`(a + (b * c)) - d`	`a b c * + d -`	`(- (+ a (* b c)) d)`
`a + ((b * c) - d)`	`a b c * d - +`	`(+ a (- (* b c) d))`
`a + (b * (c - d))`	`a b c d - * +`	`(+ a (* b (- c d)))`
`(a + b) * (c - d)`	`a b + c d - *`	`(* (+ a b) (- c d))`

Note that prefix notation is also unambiguous as long as all operators accept two arguments. Because zenlisp functions may be variadic, though, explicit operand grouping is required.

Q24, Page 129

The classes `'#[-x]` and `'#[x-]` match the characters “x” and “-” literally, because neither `x-` nor `-x` is a complete range.

The class `'#[]` matches no characters, so it is a class that never matches. `'#[^]` matches any character except for none, so it is a synonym of `'#_`. However, most real-world regex implementations would probably handle this differently.

Q25, Page 129

Removing one application of **`reverse`** is all it takes to turn the eager matcher into a lazy matcher:

```
(define (match-star cre s m)
  (letrec
    ((try-choices
       (lambda (c*)
         (cond ((null c*) :f)
               (t (let ((r (match-cre (cdr cre) (caar c*) (cadar c*))))
                    (cond (r (append (reverse m) r))
                          (t (try-choices (cdr c*))))))))))
    (try-choices (make-choices cre s ()))))
```

Make-choices returns the choices in such an order that the shortest match is listed first, so **reverse** is used in the original implementation to give precedence to the longest match. By omitting this **reverse**, the eager matcher becomes a lazy matcher.

Q26, Page 138

Evaluation of special forms cannot be delegated to the outer interpreter, because some special form handlers call **eval** internally. When this happens, the environment of the outer interpreter would be used to look up symbols that are stored in the environment of the inner interpreter.

Q27, Page 138

Because `'%special` is an ordinary symbol, the internal representation of operators can be used in programs:

```
(zeval '('(%special . quote) foo) ()) => 'foo
```

This is only a minor glitch, but you can fix it by using a unique instance [page 55] like `(list '%special)` in the place of the symbol `'%special` to tag special operators, e.g.:

```
  (let ((%special (list '%special)))
    (letrec
      ((initial-env
         (list ...
               (cons 'and    (cons %special and))
               (cons 'apply  (cons %special apply))
               ...)))
      ...))
```

An undesired effect of `'%void` is that it cannot be used as the value of a variable, because doing so would make the variable unbound:

```
(zeval 'x '((x . %void))) => bottom
```

The remedy for this effect is the same as for `'%special`.

A.7 list of figures

A.8 list of example programs

A.9 code license

Don't worry, be happy.

Frankly, life's too short to deal with legal stuff, so

– do what ever you want with the code in this book;
– if the code doesn't work, don't blame me.

index