Formica: The "functionally-oriented" language.

Version 0.0.1

Sergey B. Samoylenko

2012

Reference guide.

```
#lang formica
```

This guide describes functions and syntax forms defined in the `formica` language.

Formica is available as both a language level and a module that can be used in other languages.

1 Introduction

The *Formica* language was created for educational purpose while teaching the "Functional and logical programming" undergraduate course in the Kamchatka State Technical University (Russia). For student's practical work the *Racket* language was chosen for following reasons. *Racket* is elegant as *Scheme* and goes with educationally oriented IDE, which make newcomers feel comfortable. It has active community, reach libraries and provides a lot of real-life instruments for GUI development, web tools etc. Finally, *Racket* is extremely flexible: it encourages creating domain-oriented languages and dialects, so it is natural to come to a new language, created specially for the coursework.

The main goal of designing *Formica* is to have a functional programming language as flexible as *Racket* or *Wolfram Mathematica*, and almost as syntactically clean as Mark Tarver's *Qi* or *Haskell*. Being a dialect of *Racket* it should complement the parent language and make it possible to use any of *Racket*'s native libraries.

Even though it is mainly educational language, some of it's features (such as formal functions, rewriting, etc.) could be used in various practical applications.

This guide is addressed to a reader which is familiar both to a *Racket* and to functional programming concepts and presents only *Formica*-specific aspects.

1.1 Brief tour

Here are the main features of the Formica dialect.

1.1.1 Formal functions:

This declares `f` to be a formal function: a function which doesn't perform any calculations, but just shows it's application.

```
(define-formal f)
```

```
> (f 'x)
'(f x)
> (f 1 2 3)
'(f 1 2 3)
```

Using the formal function we can see how some high-order functions work:

```
> (map f '(a b c))
```

```
'((f a) (f b) (f c))
> (foldr f 'x0 '(a b c))
'(f a (f b (f c x0)))
> (foldl f 'x0 '(a b c))
'(f c (f b (f a x0)))
```

In the following example the '+' function is made formal:

```
> (foldr (hold +) 0 '(1 2 3))
'(+ 1 (+ 2 (+ 3 0)))
```

Together with pattern-matching and rewriting technique, formal functions give a framework for designing complex abstract data types.

1.1.2 Rewriting:

Suppose we have to define a function f that, if it receives 1 returns 0 and if it returns 0 returns 1. If all rules fail the argument is left unchanged:

```
(define f
  (rewrite 1 --> 0
           0 --> 1))

> (f 0)
1
> (f 1)
0
> (f 'x)
'x
> (f '(0 1))
'(0 1)
```

This function rewrites 1 to 2, 3 to 4, and 4 to 1 anywhere in a list.

```
(define r
  (/. 1 --> 2
      3 --> 4
      4 --> 1))

> (r '(1 2 (3 4)))
'(2 2 (4 1))
```

Following rewrites 1 to 2, 3 to 4, and 4 to 1 repeatedly, until result stops changing.

```
(define r
  (//. 1 --> 2
       3 --> 4
       4 --> 1))

> (r '(1 2 (3 4)))
'(2 2 (2 2))
```

This rewriting system has a normal form: 2.

A rewriting-based definition of the `map` function:

```
(define/. map
  _ '()        --> '()
  f (cons h t) --> (cons (f h) (map f t)))

> (map f '(a b c))
'(a b c)
```

Here is a simple implementation of symbolic η- and β-reduction rules for λ-calculus:

```
> (define//. reduce
    ; η-reduction
    `(λ. ,x (,f ,x)) --> f
    ; β-reduction
    `((λ. ,x ,B) ,A) --> (eval `((/. ',x --> ',A) ',B)))


> (reduce '((λ. x (f x x)) a))
'(f a a)
> (reduce '((λ. f (λ. x (f (f x)))) (λ. f (λ. y (f (f y))))))
'(λ. x (λ. y (x (x (x (x y))))))
```

1.1.3 Simplified syntax for partial application and point-free definitions:

Once you have written in *Haskell* on or the blackboard:

```
fold f x0 = F where F []     = x0
```

```
                    F (x:xs) = f x (fold f x0 xs)

map f = fold (cons . f) []

map (* 2) [1 2 3]
```

it is difficult not to try it in *Formica*:

```
(define/c (fold f x0) (/. '()        --> x0
                          (cons h t) --> (f h (fold f x0 t))))

(define/c (map f) (fold (∘ cons f) '()))

> (map (* 2) '(1 2 3))
'(2 4 6)
> (map (map (* 2)) '((1 2) (3 4) (5)))
'((2 4) (6 8) (10))
```

1.1.4 Contract-based dynamical typing system:

Being not so strict as in *Typed Racket* or *Haskell*, the contract typing system gives a flavor of rich type systems used in functional programming, including abstract, algebraic and polymorphic types.

```
(define-type Int-or-X
  integer?
  'X)

> (is 5 Int-or-X)
#t
> (is 'X Int-or-X)
#t
> (is 'x Int-or-X)
#f


(:: map ((a -> b) (list: a ..) -> (list: b ..))
  (define/c (map f)
    (/. (cons h t) --> (cons (f h) (map f t)))))

> (map (* 2) '(1 21 4 3))
'(2 42 8 6)
> (map cons '(1 21 4 3))
```

Signature violation: expected a procedure that accepts 1 mandatory argument without any keywords
given: #<procedure:cons>
guilty party: top-level
innocent party: function map
signature: map : (parametric->/c (a b) ...)

Building abstract algebraic types with formal functions:

A formal constructor of a pair

```
(define-formal kons)

> (kons 1 (kons 2 3))
'(kons 1 (kons 2 3))
```

A constructor of the klist

```
(define (klist . x)
  (foldr kons 'knull x))

> (klist 'a 'b 'c)
'(kons a (kons b (kons c knull)))
```

A recursive type declaration

```
(define-type klist?
  'knull
  (kons: Any klist?))

> (is (klist 1 2 3) klist?)
#t
> (is (list 1 2 3) klist?)
#f
> (is (kons 1 (kons 2 3)) klist?)
#f
```

some functions to operate with klists

```
(define/c (kfold f x0)
  (/. 'knull --> x0
```

```
      (kons h t) --> (f h (kfold f x0 t))))

(define/c (kfilter p) (kfold (fif (∘ p I1) kons I2) 'knull))

(define total (kfold + 0))

> (kfold ($ 'f) 'x0 (klist 'x 'y 'z))
'(f x (f y (f z x0)))
> (kfilter odd? (klist 1 2 3 4 5))
'(kons 1 (kons 3 (kons 5 knull)))
> (total (klist 1 2 3))
6
```

1.1.5 Monads

Monads give a very useful and elegant abstraction for sequential computations, allowing one to go deep into semantics, keeping syntax simple. Even though *Racket* doesn't need monads to perform sequential computations, side effects or set comprehension, it is useful to have monads as a powerful tool for designing new semantic constructions.

An example of sequential computations in the `List` monad:

```
> (do [x <- '(1 2 3)]
      [x <- (return (+ x 6))]
      (return x))
'(7 8 9)
```

A simple list comprehension:

```
> (collect (sqr x) [x <- (in-range 6)] (odd? x))
'(1 9 25)
```

Monads allow to define generic functions and then to use them with different semantics. Here is a function returning a sequence of Pythagorean triples:

```
(define (find-triples r)
  (collect (list a b c)
    [a <- r]
    [b <- (in-range (ceiling (/ a 2)) a)]
    [c <-: (sqrt (- (sqr a) (sqr b)))]
    (integer? c)
    (> b c)))
```

One may use `find-triples` in eager `List` monad, or in lazy `Stream` monad, without changing a single line in the function definition.

```
> (using List (find-triples (in-range 20)))
'((5 4 3) (10 8 6) (13 12 5) (15 12 9) (17 15 8))
```

```
(define t
  (using Stream (find-triples (in-naturals))))

> (stream-take t 5)
'((5 4 3) (10 8 6) (13 12 5) (15 12 9) (17 15 8))
> (stream-ref t 50)
'(100 80 60)
```

As in `Haskell`, it is possible to use pattern-matching in list-comprehensions

```
(define primitive-triples
  (using Stream
    (collect t
      [(and t (list a b _)) <- (find-triples (in-naturals))]
      [1 <-: (gcd a b)])))

> (stream-take primitive-triples 5)
'((5 4 3) (13 12 5) (17 15 8) (25 24 7) (29 21 20))
```

Consider a classical problem: *A farmer buys 100 animals for $100.00. The animals include at least one cow, one pig, and one chicken, but no other kind. If a cow costs $10.00, a pig costs $3.00, and a chicken costs $0.50, how many of each did he buy?*

Here is a declarative program solving this problem:

```
(define (solve-farmer-problem #:cow (c 10)
                              #:pig (p 3)
                              #:chicken (ch 1/2))
  (do [cows     <-  (in-range 1 (/ 100 c))]
      [pigs     <-  (in-range 1 (/ 100 p))]
      [chickens <-: (- 100 cows pigs)]
      [100 <-: (+ (* c cows) (* p pigs) (* ch chickens))]
      (return `((,cows cows) (,pigs pigs) (,chickens chickens)))))
```

```
> (solve-farmer-problem)
'(((5 cows) (1 pigs) (94 chickens)))
```

If cow costs $8 there would be 6 possible solutions:

```
> (solve-farmer-problem #:cow 8)
'(((1 cows) (17 pigs) (82 chickens))
  ((2 cows) (14 pigs) (84 chickens))
  ((3 cows) (11 pigs) (86 chickens))
  ((4 cows) (8 pigs) (88 chickens))
  ((5 cows) (5 pigs) (90 chickens))
  ((6 cows) (2 pigs) (92 chickens)))
```

It is easy to create new monads. Here is an example of defining the parameterized monad `(Maybe a)`.

```
(define-formal Maybe Just)

(define-type (Maybe? a)
  (Just: a)
  'Nothing)

(define (Maybe a)
  (monad
   #:type (Maybe? a)
   #:return (/. 'Nothing --> 'Nothing
                x       --> (Just x))
   #:bind (/. 'Nothing f --> 'Nothing
              (Just x) f --> (f x))
   #:mzero 'Nothing
   #:mplus (/. 'Nothing x --> x
               x       _ --> x)))

> (using (Maybe Int)
    (bind (Just 2) >>= (lift sqr) >>= (lift (* 2))))
'(Just 8)
> (using (Maybe Int)
    (bind 'Nothing >>= (lift sqr) >>= (lift (* 2))))
'Nothing
```

This monad could be used to perform guarded computations:

```
(define (safe-sqrt x)
```

```
  (bind (return x) >>=
        (guardf positive?) >>=
        (lift sqrt)))

> (using (Maybe Real)
    (map safe-sqrt  '(-1 4 3 9 -4 -9)))
'(Nothing (Just 2) (Just 1.7320508075688772) (Just 3) Nothing
Nothing)
> (using (Maybe Int) (safe-sqrt 3))
return: the result should have type ((Maybe? Int))
  received: '(Just 1.7320508075688772)
  using: #<monad>
```

Formica provides monadic functions and operators, such as `lift/m`, `compose/m`, `fold/m`, `map/m`, etc.

```
(using-monad List)

> (lift/m or '(#t #f) '(#t #f))
'(#t #t #t #f)
> (fold/m (λ (x y) `((+ ,x ,y) (- ,x ,y))) 0 '(1 2 3))
'((+ 3 (+ 2 (+ 1 0)))
  (- 3 (+ 2 (+ 1 0)))
  (+ 3 (- 2 (+ 1 0)))
  (- 3 (- 2 (+ 1 0)))
  (+ 3 (+ 2 (- 1 0)))
  (- 3 (+ 2 (- 1 0)))
  (+ 3 (- 2 (- 1 0)))
  (- 3 (- 2 (- 1 0))))


(define powerset (filter/m (const '(#t #f))))

> (powerset '(1 2 3))
'((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

1.1.6 Miscellaneous functional tools

Formica is informally called to be "functionally-oriented", meaning that it provides a lot of tools to operate with functions.

This defines binary and unary formal functions:

```
(define-formal (b 2) (u 1))
```

Here is a generalized composition of them:

```
> ((∘ b u) 1 2)
'(b (u 1) 2)
> ((∘ u b) 1 2)
'(u (b 1 2))
```

The generalized composition is associative and has identity function as a neutral element:

```
> (equal? ((∘ (∘ b u) b) 'a 'b 'c)
          ((∘ b (∘ u b)) 'a 'b 'c)
          (b (u (b 'a 'b)) 'c))
#t
> (equal? ((∘ id b) 1 2)
          ((∘ b id) 1 2)
          (b 1 2))
#t
```

Some purely functional definitions

```
(define max (foldr (fif < I2 I1) -inf.0))

> (max '(2 3 1 2 3 2 5 3 2))
5


(define/c (all-elements p) (foldr (∘ and p) #f))

> (all-elements odd? '(2 4 5 3 7 8))
#f
> (all-elements even? '(2 4 6 0 12 8))
#f


(define all-arguments (-< and))
(define complex<? (andf (all-arguments complex?)
                        (-< < magnitude)))
```

```
> (complex<? 0-1i 2 4+1i)
#t
> (complex<? 2 1-1i)
#f
> (complex<? 2 'x 1-1i)
#f
```

See `"examples/"` folder in the *Formica* distribution for more examples.

2 Syntax features

2.1 Partial function application

The bindings documented in this section are provided by the `formica/partial-app` library and `formica` language.

In order to get Formica language without syntax for partial application, use `"#lang formica/regular-app"` at the header of the file or `(require formica/regular-app)`. It will load all bindings from `formica` language except for those provided in `formica/partial-app` library.

One feature which makes *Formica* different from *Racket*, is simplified syntax for *partial application*, which is close to *Haskell*'s sections.

For example, function `cons`, expects two arguments:

```
> (cons 1 2)
'(1 . 2)
```

We may consider it as curried function: as a sequence of nested closures:

```
> (cons)
#<procedure:curried:cons>
> ((cons) 1)
#<procedure:curried:cons>
> (((cons) 1) 2)
'(1 . 2)
```

Here is partial application of binary function `cons`:

```
> (cons 1)
#<procedure:curried:cons>
> ((cons 1) 2)
'(1 . 2)
```

In the expression `(cons 1)` the first argument is fixed, resulting an unary function.

That's how it is possible to define the increment function:

```
> ((+ 1) 3)
4
> (define inc (+ 1))
```

```
> (inc 3)
4
> (map (+ 1) '(1 2 3))
'(2 3 4)
```

The simplified syntax makes possible only "left" sections by fixing arguments from left to right. For fixing the sequence of arguments "from the right" one has to use explicit partial application using `curryr` function.

Examples of explicit partial application:

```
> (- 1)
-1
> (curry - 1)
#<procedure:curried:->
> ((curry - 1) 3)
-2
> (map (curry - 1) '(1 2 3))
'(0 -1 -2)
> (curryr - 1)
#<procedure:curried:->
> ((curryr - 1) 3)
2
> (map (curryr - 1) '(1 2 3))
'(0 1 2)
```

(apply *f* *v* ...) → Any
  f : Fun
  v : Any

Applies function *f* to a list of arguments *v* .... If the number of arguments is less then arity of function *f* returns partially applied function.

```
> (apply cons '(1 2))
'(1 . 2)
> (apply cons '(1))
#<procedure:curried:cons>
> (apply cons '(1 2 3))
cons: arity mismatch;
 the expected number of arguments does not match the given
number
  expected: 2
  given: 3
```

```
arguments...:
 1
 2
 3
```

The syntax for partial application may make compilation time of a program be longer then usual. However, usually it does not affect the run-time efficiency.

2.2 Point-free function definitions

The bindings documented in this section are provided by the `formica/tacit` library and `formica` language.

```
(define/c (f var ...) rhs)

            rhs = fun-constructor
                | (g args ...)
                | lit

fun-constructor = (lambda arg-spec body)
                | (match-lambda arg-spec body)
                | (case-lambda arg-spec body)
                | (rewrite rules ...)
                | (rewrite-all rules ...)
                | (rewrite-repeated rules ...)
                | (rewrite-all-repeated rules ...)
                | (//. rules ...)
                | (/. rules ...)
```

Defines a function *f* in the *point-free notation*. The right-hand side of the definition *rhs* can be either a function constructor *fun-constructor*, partially applied function *g* with fixed arguments *args ...*, or any literal expression *lit*. The defined function *f* has formal arguments *var ...*, complemented by free arguments of the right right-hand side *rhs*, if any.

Examples:

```
> (define/c (F x)
    (case-lambda
      [(y) `(F ,y ,x)]
      [(y z) `(F ,x ,y ,z)]))
```

```
> (procedure-arity F)
'(2 3)
> (F 1 2)
'(F 2 1)
> (F 1 2 3)
'(F 1 2 3)
```

```
> (define/c (map1 f)
    (/. (cons h t) --> (cons (f h) (map1 f t))))

> (procedure-arity map1)
2
> (map1 ($ 'f) '(a b c))
'((f a) (f b) (f c))
```

```
> (define/c (map2 f) (foldr (∘ cons f) '()))

> (procedure-arity map2)
(arity-at-least 2)
> (map2 ($ 'f) '(a b c))
'((f a) (f b) (f c))
> (map2 ($ 'g 2) '(a b c) '(x y z))
'((g a x) (g b y) (g c z))
```

```
> (define/c (f x) `(,x ,x))

> (procedure-arity f)
1
> (f 'a)
'(a a)
> (f '(a b c))
'((a b c) (a b c))
```

3 The Term Rewriting

The bindings documented in this section are provided by the `formica/rewrite` library and `formica` language.

The Formica provides tools for programming via term rewriting technique. Provided forms could be considered as a syntactic sugar for Racket's `match` form, however they offer different semantics. The language introduces repetitive rewriting in order to obtain the normal form of given expression, ability to transform any sub-part in nested list structure and definition of formal functions which abstract general algebraic types.

3.1 Motivation

The thoroughly developed rewriting theory has many applications in different fields of computer science [Baader99, Bezem03]. It could be a handy tool in everyday programming practice, making program units shorter and more expressive. The REFAL [Turchin89, Surhone10] and Wolfram's Mathematica core language [Wolfram] give excellent examples of scalability, power and expressiveness provided by rewriting.

The aim of developing this library is to mimic Mathematica's rewriting tools in Scheme. The library provides four forms: `rewrite`, `rewrite-all`, `rewrite-repeated` and `rewrite-all-repeated` named almost in the same way as rewriting functions in Mathematica. In spite of this naming convention, we use word "rewriting" for "replacement", "substitution" or "transformation" in this manual, since "rewriting" has precise mathematical meaning.

Rewriting is a formal method of replacing subterms of an expression with other terms. In Formica the role of the expression play lists, structures and formal applications.

3.1.1 Short examples

Rewriting is a function:

```
> (/. 'a --> 'b)
#<procedure:rewrite-all>
```

It replaces all occurrences of `'a` to `'b` in a nested list structure:

```
> ((/. 'a --> 'b) '(1 a ((a b c) a)))
'(1 b ((b b c) b))
```

This rewriting describes cyclic swapping of symbols:

```
> ((/. 'a --> 'b 'b --> 'c 'c --> 'a) '(1 a ((a b c) a)))
'(1 b ((b c a) b))
```

Rewriting rules used as lambda-functions:

```
> ((/. x --> (* x x)) 4)
16
> ((/. x y --> (* y x)) 4 5)
20
> ((/. x (cons y z) --> (values (* x y) z)) 4 '(1 2 3))
4
'(2 3)
```

Rewriting is done at any level of a list structure:

```
> ((/. (? number? x) --> (* x x)) '(1 (2 3) 4))
'(1 (4 9) 16)
```

Rewriting rules accept any pattern recognized by `match` form:

```
> ((/. (list _ (and x (app length 2)) (or 1 2)) --> x) `(a (1 4) 2))
'(1 4)
```

Repetitive rewriting could be used to find a fixed point of a function. This finds a square root of 2 using Newton's method:

```
> ((//. x  --> (/ (+ x (/ 2 x)) 2)) 1.0)
1.4142135623730950
```

Simple implementation of symbolic η- and β-reduction rules for λ-calculus:

We need `eval` here because neither rewriting rules nor replace form (`/.`) are first class objects (same as `match` and it's patterns).

```
> (define reduce
    (//. ; η-reduction
         `(λ. ,x (,f ,x)) --> f
         ; β-reduction
         `((λ. ,x ,B) ,A) --> (eval `((/. ',x --> ',A) ',B))))
```

These rules read: "The λ-term $\lambda.x\ (f\ x)$ could be rewritten as f ". "Application of a λ-term $((\lambda\ .\ x\ B)\ A)$ is done by rewriting each occasion of a formal argument x by A in the function's body B". Now we are able to make symbolic reduction of λ-terms:

```
> (reduce '((λ. x (f x x)) a))
'(f a a)
> (reduce '((λ. f (λ. x (f (f x)))) (λ. f (λ. y (f (f y))))))
'(λ. x (λ. y (x (x (x (x y))))))
> (define ω '(λ. f (λ. x ((f f) x))))

> (reduce `(,ω (λ. x x)))
'(λ. x x)
> (reduce `((λ. x y) (,ω ,ω)))
'y
```

To be precise, it is not really lazy. If the whole expression were not a β-redex, rewriting would proceed with it's parts and finally fall into $(\omega\ \omega)$ infinite loop.

The last example shows that we have implemented a sort of lazy evaluation. This is a reflection of the rewriting strategy: first try to rewrite the whole expression and then it's parts.

More examples could be found in the `"formica/examples"` folder within the package distribution. Examples include:

- `"rewrite.rkt"` – more basic examples.
- `"automata.rkt"` – finite automata defined with rewriting.
- `"infix.rkt"` – transformation of algebraic expressions given in infix notation, to prefix notation with parenthesis and reverse polish notation.
- `"peano.rkt"` – definition of Peano axioms.
- `"logics.rkt"` – simple tautology prover.
- `"turing.rkt"` – a Turing machine interpreter.

3.2 Rewriting Forms

3.2.1 Singlefold Rewriting

```
(rewrite rule-spec ...)

rule-spec = (pat ... --> expr)
          | (pat ... --> (=> id) expr)
          | (pat ... --> (? guard) expr)
```

Transforms to a function which applies rules specified by *rule-spec* in an attempt to transform the entire input expression. If no rules could be applied, input expression is left unchanged.

pat — any pattern suitable for the `match` form. The sequence of patterns correspond to a sequence of arguments to which a rewriting function is applied.

expr — any expression. In contrast to `match` the right-hand side part of the rule should contain only one expression. Use `begin` for sequencing.

Examples:

```
(define f
  (rewrite x --> (* x x)
           x y --> (* x y)))

> (f 4)
16
> (f 4 5)
20
> (procedure-arity f)
'(1 2)
```

If none of patterns match, the input arguments are returned as a list.

An optional `(=> id)` between patterns and the *expr* is bound to a failure procedure of zero arguments. It works in the same way as in the `match` form.

An optional `(? guard)` between patterns and the *expr* is used to guard the rule. The *guard* is evaluated with bindings given by patterns. If *guard* evaluates to `#f` evaluation process escapes back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. Otherwise the rule is applied.

Examples:

```
> (define g
    (rewrite x y --> (? (< x y)) (* x y)
             _ y --> y))

> (g 3 4)
12
> (g 4 3)
3
```

```
(rewrite-all rule-spec ...)
```

Transforms to a procedure which applies rules specified by *rule-spec* in an attempt to transform each subpart of the input expression. If no rules could be applied, input expression is left unchanged.

```
(/. rule-spec ...)
```

An alias for the `rewrite-all` form.

Examples:

```
> ((/. 'a --> 'x) '(a b a d a))
'(x b x d x)
> ((/. 'a --> 'x) '(a (b (a) d) a))
'(x (b (x) d) x)
> ((/. 'a --> 'b 'b --> 'a) '(a (b (a) b) a))
'(b (a (b) a) b)
```

Only unary rules could be applied to subexpressions. At the same time multiary rules could be applied to a sequence of input arguments.

Examples:

```
> (define g
    (/. (? number? x) --> (* x x)
        x y --> (* x y)))

> (g 4)
16
> (g '(4 5))
'(16 25)
> (g 4 5)
20
```

```
(define/. id rule-spec ...)
```

Expands to `(define id (/. rule-spec ...))`.

3.2.2 Repetitive Rewriting

Repetitive rewriting rules could be either *regular* or *terminal*. Application of repetitive rewriting rules follows the algorithm:

- Consequently try to apply given rules to the expression.
- If a pattern, corresponding to a *regular* rule matches (rule with `-->` arrow), make rewriting and apply rules to the result, starting from the beginning of the rule sequence.
- If a pattern, corresponding to a *terminal* rule matches (rule with `-->.` arrow), make rewriting, stop the rewriting process and return the result.
- The rewriting process stops either if expression does not change after rewriting, or if the last applied rule was terminal.

```
(rewrite-repeated rule-spec ...)

rule-spec = (pat ... ar expr)
          | (pat ... ar (=> id) expr)
          | (pat ... ar (? guard) expr)

       ar = -->
          | -->.
```

Transforms to a function which applies rules, specified by *rule-spec*, repeatedly in an attempt to get the normal form of the entire input expression.

Arrows `-->` and `-->.` correspond to regular and terminal rewriting rules, respectively.

```
(rewrite-all-repeated rule-spec ...)
```

Transforms to a procedure which repeatedly performs rewriting in every part of the given expression until either result no longer changes, or terminal rule is applied. It performs one complete pass over the expression using `rewrite-all`, then carries out the next pass. Each pass is done using preorder traversal of nested list structure.

```
(//. rule-spec ...)
```

An alias for `rewrite-all-repeated` form.

Example:

```
> ((//. 'a --> 'b
        'b --> 'c
        'c --> 'd) '(a b c))
'(d d d)
```

Using a terminal rule

```
> ((//. 'a -->. 'b
        'b --> 'c
        'c --> 'a) '(a b c))
'(b b b)
```

If rewriting proceeds with multiple values, these values could be accepted as arguments during repeating iterations.

Example. Calculation of GCD:

```
> ((//. a 0 -->. a
        a b --> (values b (modulo a b))) 225 125)
25
```

Let's compare three functions which iteratively calculate the n-th Fibonacci number. The first of them is defined in a regular recursive way, the second implements recursion via rewriting, and the third one uses repetitive rewriting.

```
(define (fib1 n)
  (case n
    [(1) 0]
    [(2) 1]
    [else (let loop ([a 0] [b 1] [i n])
            (if (= i 3)
              (+ a b)
              (loop b (+ a b) (- i 1))))]))

(define fib2
  (rewrite
   1 --> 0
   2 --> 1
   n --> (fib2 0 1 n)
   a b 3 --> (+ a b)
   a b i --> (fib2 b (+ a b) (- i 1))))

(define fib3
  (rewrite-repeated
   1 -->. 0
   2 -->. 1
   n --> (values 0 1 n)
   a b 3 -->. (+ a b)
   a b i --> (values b (+ a b) (- i 1))))

> (let ([n 10000])
    (= (time (fib1 n)) (time (fib2 n)) (time (fib3 n))))
cpu time: 48 real time: 51 gc time: 0
cpu time: 40 real time: 50 gc time: 0
cpu time: 84 real time: 122 gc time: 32
#t
```

`(define//. id rule-spec ...)`

Expands to `(define id (//. rule-spec ...))`.

4 Formal functions

The bindings documented in this section are provided by the `formica/formal` library and `formica` language.

The concept of formal functions was taken from symbolic computation languages, such as *Wolfram Mathematica* or *Maxima*. Together with pattern-matching and rewriting techniques formal functions give powerful and flexible framework for development of abstract data types, symbolic computations, debugging of functional programs and teaching different programming concepts.

A *formal function* if applied to any arguments does not provide any computations and returns a literal expression denoting it's application (the *formal application*).

```
(f 'x 'y) ==> '(f x y)
```

The formal application is just a list, having the name of function at the first position and the sequence of arguments as the rest of it.

Formal functions are in many ways similar to Racket's structures: they provide tagged containers, which could be identified and analyzed by pattern-matching and contract systems. Here are some points which show the difference between formal functions and structures.

- Formal functions do not give names to their arguments as Racket structures do.
- Unlike structure formal function may accept different number of arguments, as a variadic or polyadic function. Structures allowed to have only optional fields with default values.
- Because formal applications are lists, they are much more flexible then structures. Formal applications could be combined and transformed, mapped and folded as any list. However they could be identified by pattern-matching, as structures.

4.1 Motivation

Pattern matching usually goes together with algebraic types. Racket's structures and lists could be used as constructors for algebraic data types, but they use have some drawbacks.

Suppose we wish to define a rewriting system for Peano axioms. We start with definition of types for numerals, successor, sum and product:

```
(struct N (x) #:transparent)
(struct 1+ (x) #:transparent)
```

```
(struct Sum (x y) #:transparent)
(struct Prod (x y) #:transparent)
```

Next we define rewriting rules:

```
(define calculate
  (rewrite-all-repeated
   (N 0) --> 0
   (N n) --> (1+ (N (- n 1)))

   (Sum x 0) --> x
   (Sum x (1+ y)) --> (1+ (Sum x y))

   (Prod _ 0) --> 0
   (Prod x (1+ y)) --> (Sum x (Prod x y))))

> (calculate (Sum (N 2) (N 3)))
(Sum (N 2) (N 3))
```

However, this clear definition doesn't work, because the rewriting process can't get into the structures and change their parts. One solution is to add rules for penetrating into the structures:

```
(define calculate2
  (rewrite-repeated
   (N 0) --> 0
   (N n) --> (1+ (N (- n 1)))

   (Sum x 0) --> x
   (Sum x (1+ y)) --> (1+ (Sum x y))

   (Prod _ 0) --> 0
   (Prod x (1+ y)) --> (Sum x (Prod x y))

   (1+ x) --> (1+ (calculate2 x))
   (Sum x y) --> (Sum (calculate2 x) (calculate2 y))
   (Prod x y) --> (Prod (calculate2 x) (calculate2 y))))

> (calculate2 (Sum (N 2) (N 3)))
(1+ (1+ (1+ (1+ (1+ 0)))))
> (calculate2 (Prod (N 2) (N 3)))
(1+ (1+ (1+ (1+ (1+ (1+ 0))))))
```

Now it works! But heavy and tautological lines which end up the rewriting system look cumbersome. They describe programming, not Peano axioms.

Moreover, we need explicit recursion, therefore we have lost the ability to make anonymous rewriting system. Finally, use of structures fixes the number of their fields, so it is impossible to write a pattern like `(Sum x __)` and to have any number of arguments, which is reasonable when dealing with sums.

Another approach is to use lists and interpret their `car`-s as tags.

```
(define calculate3
  (rewrite-all-repeated
   `(N 0) --> 0
   `(N ,n) --> `(1+ (N ,(- n 1)))

   `(Sum ,x 0) --> x
   `(Sum ,x (1+ ,y)) --> `(1+ (Sum ,x ,y))

   `(Prod ,_ 0) --> 0
   `(Prod ,x (1+ ,y)) --> `(Sum ,x (Prod ,x ,y))))

> (calculate3 '(Sum (N 2) (N 3)))
'(1+ (1+ (1+ (1+ (1+ 0)))))
> (calculate3 '(Prod (N 2) (N 3)))
'(1+ (1+ (1+ (1+ (1+ (1+ 0))))))
```

Looks much better, but a bit prickly because of quotes and commas. It may become even worse-looking if we use blanks _ and ___ a lot. Using explicit `list` constructor does not help either, it makes patterns more difficult to read and write: `(list 'Sum x (list '1+ y))`.

The package provides an abstraction named *formal function* which work as a tagged list constructor in expression position and as a structure constructor in patterns. With use of formal functions we may write clear definition of rewriting system for Peano's numerals without a single redundant word or symbol:

```
(define-formal N 1+ Sum Prod)
(define calculate4
  (rewrite-all-repeated
   (N 0) --> 0
   (N n) --> (1+ (N (- n 1)))

   (Sum x 0) --> x
   (Sum x (1+ y)) --> (1+ (Sum x y))
```

```
    (Prod _ 0) --> 0
    (Prod x (1+ y)) --> (Sum x (Prod x y))))

> (calculate4 (Sum (N 2) (N 3)))
'(1+ (1+ (1+ (1+ (1+ 0)))))
> (calculate4 (Prod (N 2) (N 3)))
'(1+ (1+ (1+ (1+ (1+ (1+ 0))))))
```

Besides, formal functions could be useful in general. For example in exploring functional programming:

```
> (define-formal f g h)

> (map f '(a b c))
'((f a) (f b) (f c))
> (foldl g 'x0 '(a b c))
'(g c (g b (g a x0)))
> (foldr g 'x0 '(a b c))
'(g a (g b (g c x0)))
> ((compose f g h) 'x 'y)
'(f (g (h x y)))
```

4.2 Creation of formal functions

```
(define-formal f-spec ...)

f-spec = id
       | (id arity-spec)

  id : symbol?

  arity-spec : procedure-arity?
```

Defines formal functions, according to specifications *f-spec ...*, where *id* is the name of a formal function and *arity-spec* is it's arity.

For each formal function *id* the `define-formal` form also defines

- the predicate *id*`?`, which returns `#t` for formal application of *id* and `#f` otherwise;
- the syntax form *id*`:`, which could be used as a contract for the formal application with arguments of given types.

Examples:

```
> (define-formal f)

> f
#<procedure:formal:f>
> (f 2)
'(f 2)
> (f 'x 'y)
'(f x y)
> (map f '(a b c) '(x y z))
'((f a x) (f b y) (f c z))
> (foldr f 'x0 '(a b c))
'(f a (f b (f c x0)))
> (f? (f 1 2))
#t
> (f? '(1 2))
#f
> (is (f 1) (f: Num))
#t
> (is (f 1 'x) (f: Num Sym))
#t
> (is (f 1 2 3) (f: Num ..))
#t
```

Formal function with fixed arity:

```
> (define-formal (g 2))

> g
#<procedure:formal:g>
> (g 2)
#<procedure:curried:formal:g>
> (g 'x 'y)
'(g x y)
> (g 1 2 3)
formal:g: arity mismatch;
  the expected number of arguments does not match the given
number
    expected: 2
    given: 3
    arguments...:
      1
      2
      3
> (map g '(a b c) '(x y z))
'((g a x) (g b y) (g c z))
```

Using formal applications as patterns:

```
> (define-formal f (g 2))

> (match (f 5)
     [(f x) x])
5
> (match (f 1 2 3 4)
     [(f x y ...) y])
'(2 3 4)
> (match (g 1 (f 2 3))
     [(g a (f b c)) (list a b c)])
'(1 2 3)
> (match (g 1 2)
    [(g a) a])
match: no matching clause for '(g 1 2)
> (match (g 1 2)
    [(g a b c) (list a b c)])
match: no matching clause for '(g 1 2)
```

The last two cases have wrong syntax because `g` was declared to be a binary function.

```
(hold f [arity]) → formal-function?
  f : (or/c Fun Sym)
  arity : procedure-arity? = (arity-at-least 0)
$ : hold
```

Returns a formal function, named like function *f* with arity specified by *arity*. Function `hold` has an alias: `$`.

Examples:

```
> (hold +)
#<procedure:formal:+>
> (map + '(1 2 3) '(10 20 30))
'(11 22 33)
> (map ($ +) '(1 2 3) '(10 20 30))
'((+ 1 10) (+ 2 20) (+ 3 30))
> (foldr ($ +) 'x0 '(a b c))
'(+ a (+ b (+ c x0)))


> ($ 'f 2)
#<procedure:formal:f>
> (($ 'f 2) 1)
```

```
#<procedure:curried:formal:f>
> ((($ 'f 2) 1) 2)
'(f 1 2)
> (($ 'f 2) 1 2)
'(f 1 2)
> (($ 'f 2) 1 2 3)
formal:f: arity mismatch;
 the expected number of arguments does not match the given
number
  expected: 2
  given: 3
  arguments...:
   1
   2
   3
```

```
(formal-function? x) → Bool
  x : Any
```

Returns `#t`, if `x` is a formal function, and `#f` otherwise.

Examples:

```
> (define-formal f)

> (formal-function? f)
#t
> (formal-function? +)
#f
> (formal-function? (hold +))
#t
```

4.3 Identification of formal applications

```
(formal? x) → Bool
  x : Any
```

Returns `#t`, if `x` is a formal application, and `#f` otherwise. This predicate distinguishes only applications of formal functions, defined using `define-formal`.

Examples:

```
> (define-formal f)

> (formal? (f 1 2))
#t
> (formal? '(+ 1 2))
#f
> (formal? (($ +) 1 2))
#f
```

```
(formal patt)
```

A pattern which matches any formal application. It allows to name the head of a formal application, not only arguments.

Examples:

```
> (define-formal f)

> (match (f 1 2)
     [(formal (F args ...)) `(function: ,F arguments: ,args)])
'(function: f arguments: (1 2))
> (match '(+ 1 2)
     [(formal (F args ...)) `(function: ,F arguments: ,args)])
match: no matching clause for '(+ 1 2)
```

```
(n/f-pair? x) → Bool
  x : Any
(n/f-list? x) → Bool
  x : Any
```

Return `#t`, if *x* is a pair (list) but not a formal application, and `#f` otherwise.

Examples:

```
> (define-formal f)

> (n/f-pair? (f 1 2))
#f
> (n/f-pair? '(F 1 2))
#t
> (n/f-list? (f 1 2))
#f
> (n/f-list? '(F 1 2))
#t
```

```
(formal-out id ...)
```

A provide transformer which for each name of the formal function *id* provides the predicate *id?* and form *id:*.

Examples:

```
> (module a formica
    (define-formal h)
    (provide (formal-out h)))

> (h 1 2)
'(h 1 2)
> (require 'a)

> (h 1 2)
'(h 1 2)
> (h? (h 1 2))
#t
> (formal? (h 'x 'y))
#t
> (match (h 1 2 3 4)
     [(h x y ...) y])
'(2 3 4)
```

5 The contract-based type system

The bindings documented in this section are provided by the `formica/types` library and `formica` language.

Formica has *strict dynamic type system*, with type checking being done at run-time.

Types are used for identification and guarding only. There is no optimisation based on types as in Typed Racket. It is not necessary to declare types of functions and arguments, they are used in the same way as Racket contracts.

Contract-based type system is implemented in Formica for educational purpose: in order to give students a gentle but quite deep introduction to type systems used in functional programming languages. In Formica it is possible to declare and use abstract algebraic, inductive, parametrized and functional types, still being able to utilize rewriting, simplified syntax for partial application and definitions in point-free notation. Moreover the Formica types have close relation to a concept of formal functions widely used in the language.

The type in Formica is bound to a value, not to a variable, it represents a set to which value belongs. Such a set could be specified by enumeration, a predicate function or by the inductive definition.

In Formica any type can be either

- a primitive type,
- an algebraic type,
- a functional type or a function signature.

The type checking could be done by conditional forms: `if`, `cond`, `when` etc., or using function signature.

5.1 Contracts

All types are defined by *contracts*. A role of a contract could play

- a constant which belongs to a primitive type;
- an unary predicate, describing the type;
- a compound contract, constructed by contract combinators.

```
(contract? v) → Bool
  v : Any
```

Returns `#t` if *v* could be used as a contract and `#f` otherwise.

Any constant which belongs to a primitive type could be used as a contract, representing a *unit type*:

```
> (contract? 5)
#t
> (contract? 'abc)
#t
```

Any predicate could be used as a contract:

```
> (contract? number?)
#t
> (contract? procedure-arity?)
#t
> (contract? cons)
#f
```

Contracts could be constructed using contract combinators:

```
> (contract? (and/c integer? positive?))
#t
> (contract? (or/c number? (cons: number? number?)))
#t
```

```
(is v type-pred)

  v : Any
  type-pred : contract?
```

Provides safe type check. Returns `#t` if *v* belongs to a type, defined by the contract *type-pred*, and `#f` otherwise.

This form differs from direct contract application in following:

- it allows to consider primitive types;
- if application of *type-pred* leads to exception, the `is` form does not stop running the program and returns `#f`.

```
> (is 'abc symbol?)
#t
> (is 'abc 'abc)
#t
> (is 1.2 odd?)
#f
```

Direct application of `odd?` predicate to non-integer value leads to an error:

```
> (odd? 1.2)
odd?: contract violation
  expected: integer
  given: 1.2
```

Any value may belong to unlimited number of types. For example number `5` belongs to:

- the unit type `5`:

  ```
  > (is 5 5)
  #t
  ```

- the numeric type:

  ```
  > (is 5 number?)
  #t
  ```

- the integer number type:

  ```
  > (is 5 integer?)
  #t
  ```

- the numeric type "odd number":

  ```
  > (is 5 odd?)
  #t
  ```

- algebraic type:

  ```
  > (is 5 (or/c 0 1 2 3 5 8 13))
  #t
  ```

  and so on.

5.2 Primitive data types

A value belongs to a *primitive type* if it belongs either to

- a data type, defined by corresponding predicate (`boolean?`, `number?`, `real?`, `integer?`, `positive?`, `string?`, `symbol?` etc.),
- or to a *functional type*.

All functions and rewriting systems satisfy the `function?` predicate.

A value belongs to a *functional type* if it belongs to

- memoized functions, defined by `memoized?` predicate;
- curried or partially applied functions, defined by `curried?` predicate;
- formal functions, defined by `formal-function?` predicate;
- contracts, defined by `contract?` predicate.

Functional types which specify types for arguments and result of a function could be declared using function signatures.

Some frequently used primitive types have short names, which denote sets, defined by predicates. They could be used in type definitions, signatures and contracts.

```
Bool : contract?
```

defines a set of boolean values. Equivalent to `boolean?`.

```
Num : contract?
```

defines a set of numeric values. Equivalent to `number?`.

```
Real : contract?
```

defines a set of real numbers. Equivalent to `real?`.

```
Int : contract?
```

defines a set of integer numbers. Equivalent to `integer?`.

```
Nat : contract?
```

defines a set of natural numbers.

```
Index : contract?
```

defines a set of positive integer exceeding zero.

```
Str : contract?
```

defines a set of strings. Equivalent to `string?`.

```
Sym : contract?
```

defines a set of symbols. Equivalent to `symbol?`.

```
Fun : contract?
```

defines a set of functions. Equivalent to `function?`.

5.3 Defining new types

```
(define-type name c ...)
(define-type (name x ...) c ...)

  c : contract?
  x : contract?
```

Defines named contract which returns `#t`, if the argument satisfies any contract within *c ...*. If parameters *x ...* are given, defines the contract for parametrized type.

Types defined by `define-type` represent *algebraic types*, where the sequence *c ...* represents the *type sum*, container types correspond to *type products* and primitive types correspond to *unit types*.

5.3.1 Contract combinators

Contract combinators allow to construct new types out of existing primitive types or container types.

```
(Any v) → contract?
  v : Any
```

The contract for any value.

```
(or/c c ...) → contract?
  c : contract?
(and/c c ...) → contract?
  c : contract?
(not/c c) → contract?
  c : contract?
```

Union, intersection and negation of contracts.

5.3.2 Container types

Algebraic data types could be created using *container types*: pairs, lists, formal functions or structures.

```
(cons: c1 c2) → contract?
  c1 : contract?
  c2 : contract?
```

The contract for a pair of values, which belong to types *c1* and *c2*.

```
> (is (cons 1 2) (cons: 1 2))
#t
> (is (cons 1 2) (cons: Num Num))
#t
> (is (cons 1 'x) (cons: Num Num))
#f
> (is (cons 1 (cons 2 'x)) (cons: Num (cons: Num Sym)))
#t
```

```
(list: c ...)
(list: c ..)

  c : contract?
```

The contract for lists.

- `(list: c ...)` contract for a list with fixed number of elements. The number of elements must be equal to a number of contracts *c ...* and all elements must satisfy corresponding contracts.

```
> (is '(1 2) (list: 1 2))
#t
> (is '(1 2) (list: Num Num))
#t
> (is '(1 2) (list: Num Sym))
#f
> (is '(1 2 -3) (list: positive? positive? negative?))
#t
```

- `(list: c ..)` contract for list of elements, satisfying contract `c`.

```
> (is '(1 2) (list: Num ..))
#t
> (is '(1 1 1 1) (list: 1 ..))
#t
> (is '(1 2 x 4 5) (list: Num ..))
#f
> (is '(1 2 30 1) (list: positive? ..))
#t
```

```
(f: c ...)

  c : contract?
```

contract for the formal application of function *f*. Similar to `list:` combinator.

All contracts defined in the `racket/contract` module also could be used.

Example 1.

Special types could be defined as usual predicate functions:

```
(define-type Nat
  (andf integer? positive?))

> (is 9 Nat)
#t
> (is -4 Nat)
#f
> (is "abc" Nat)
#f
```

To deal with unit types use contract combinators:

```
(define-type Nat*
  (and/c (not/c 4) integer? positive?))

> (is 3 Nat*)
#t
> (is 4 Nat*)
#f
> (is 5 Nat*)
#t
```

Example 2.

Enumerable type "friend":

```
(define-type Friend
  "Andrew" "Mary" "Park Hae Dong" "James")

> (is "Mary" Friend)
#t
> (is "Gregory" Friend)
#f
> (is 845 Friend)
#f
```

Example 3.

Inductive type "List":

```
(define-type List
  null?
  (cons: Any List))

> (is '(a b c) List)
#t
> (is (cons 'a (cons 'b (cons 'c '()))) List)
#t
> (is (cons 'a (cons 'b 'c)) List)
#f
> (is 42 List)
#f
```

Example 4.

Inductive parametrized type: "List of A":

```
(define-type (Listof A)
  null?
  (cons: A (Listof A)))

> (is '(1 2 3) (Listof Int))
#t
> (is '(1 2 x) (Listof Int))
#f
> (is '(a b c) (Listof Sym))
#t
```

Example 5.

This defines an abstract data type equivalent to a `cons`-pair:

```
(define-formal kons)

> (kons 1 2)
'(kons 1 2)
> (kons (kons 1 2) (kons 3 4))
'(kons (kons 1 2) (kons 3 4))


> (is (kons 1 2) kons?)
#t
> (is (cons 1 2) kons?)
#f
> (is 42 kons?)
#f
```

Here is a type definition and the constructor of lists, based on `kons`-pairs. We use the `'knull` symbol as an empty list: `klist`s:

```
(define-type klist?
  'knull
  (kons: Any klist?))

> (is (kons 1 (kons 2 'knull)) klist?)
#t
> (is (kons 1 (kons 2 3)) klist?)
#f
> (is (list 1 2 3) klist?)
#f
```

```
(define (klist . x)
  (foldr kons 'knull x))

> (klist 1 2 3 4)
'(kons 1 (kons 2 (kons 3 (kons 4 knull))))
> (is (klist 1 2 3) klist?)
#t
> (is (klist) klist?)
#t
```

Finally let's define folding of `klist`s and it's ancestors:

```
(define/c (kfold f x0)
  (/. 'knull --> x0
      (kons h t) --> (f h (kfold f x0 t))
      x --> (error "kfold: The argument must be a klist.
Given" x)))



(define/c (kmap f) (kfold (o kons f) 'knull))
(define total (kfold + 0))

> (kmap sqr (klist 1 2 3 4 5))
'(kons 1 (kons 4 (kons 9 (kons 16 (kons 25 knull)))))
> (total (klist 1 2 3 4 5))
15
> (kmap sqr (list 1 2 3 4 5))
kfold: The argument must be a klist. Given (1 2 3 4 5)
```

Example 6.

Let's define a parametrized abstract algebraic type to represent a binary tree having leaves of given type:

```
(Tree A) = Empty
         | (Leaf A)
         | (Node (Tree A) (Tree A))
```

As a type constructors `Leaf` and `Node` we will use formal functions, the unit type `Empty` will be represented by a symbol `'Empty`.

```
(define-formal Leaf Node)
```

```
(define-type (Tree A)
  'Empty
  (Leaf: A)
  (Node: (Tree A) (Tree A)))
```

Now we are able to create binary trees:

```
(define A (Node (Node (Leaf 5)
                      'Empty)
                (Node (Node 'Empty
                            (Leaf 6))
                      (Leaf 2))))
(define B (Node (Leaf 'a)
                (Node (Leaf 'b)
                      (Node (Leaf 'c)
                            'Empty)))))
```

```
> (is A (Tree Int))
#t
> (is A (Tree Sym))
#f
> (is B (Tree Sym))
#t
```

Let's define folding function for binary trees:

```
(define/c (tfold f x0 g)
  (/. 'Empty --> x0
      (Leaf x) --> (g x)
      (Node l r) --> (f (tfold f x0 g l)
                        (tfold f x0 g r))
      x --> (error "The argument must be a Tree. Given" x)))
```

Using folding it is possible to define a number of different functions:

Mapping:

```
(define/c (tmap f) (tfold Node 'Empty (o Leaf f)))
```

```
> (tmap sqr A)
'(Node (Node (Leaf 25) Empty) (Node (Node Empty (Leaf 36)) (Leaf
4)))
```

List of leaves

```
(define leaves (tfold append '() list))

> (leaves A)
'(5 6 2)
> (leaves B)
'(a b c)
> (leaves '(1 2 3))
The argument must be a Tree. Given (1 2 3)
```

Sum of elements (for numeric entries):

```
(define (total t)
  (cond
    [(is t (Tree Num)) (tfold + 0 id t)]
    [else (error "total: The argument must be a (Tree number?).
Given" t)]))

> (total A)
13
> (total B)
total: The argument must be a (Tree number?). Given (Node
(Leaf a) (Node (Leaf b) (Node (Leaf c) Empty)))
```

5.4 Function signatures

The *function signature* defines the domain and range of a function. On the other hand the signature plays role of the contract: it defines preconditions and postconditions for a function in terms of predicates which should be satisfied by arguments and a result. In case of pure functions this two interpretations of signature coincide.

```
(-> dom ... range)
(-> dom ... rest .. range)
(-> dom ... (? opt ...) range)
(-> dom ... (? opt ...) rest .. range)
```

```
dom : contract?
range : contract?
opt : contract?
rest : contract?
```

Defines a function signature, having domain *dom ...* and range *range*. Function may have optional arguments *opt ...*. For variadic functions the type of arguments is given by the contract *rest* followed by symbol `..` .

Could be used in the infix notation:

```
(dom ... -> range)
(dom ... rest .. -> range)
(dom ... (? opt ...) -> range)
(dom ... (? opt ...) rest .. -> range)
```

```
(:: f sig fun-def)
```

Binds the signature *sig* with the function *f*, defined by *fun-def*. Signature may include unbounded symbols which are interpreted as polymorphic types.

The definition *fun-def* could be given using any binding form: `define`, `define/c`, `define/.`, `define//.`, `define/memo` or `define/memo/c`.

Examples

Definition of numeric function:

```
(:: sqr (Num -> Num)
 (define (sqr x)
   (* x x)))

> (sqr 3)
9
> (sqr 'a)

  Signature violation:  expected: Num
 given: 'a
  guilty party:  top-level
  innocent party:  function sqr
  signature:  sqr : (Num -> Num)
```

This defines a function, which must have a positive number and a symbol as first and second arguments, resulting an even number.

```
(:: f (positive? Sym -> even?)
 (define (f x s)
   (* 2 x)))

> (f 3 'a)
6
> (f -3 'a)

  Signature violation:  expected: positive?
 given: -3
  guilty party:  top-level
  innocent party:  function f
  signature:  f : (positive? Sym -> even?)
> (f 3/2 'a)

  Signature violation:  promised: even?
 produced: 3
  guilty party:  function f
  innocent party:  top-level
  signature:  f : (positive? Sym -> even?)
```

The signature of variadic function:

```
(:: append (list? list? .. -> list?)
 (define/. append
   a --> a
   a '() --> a
   a b --> (foldr cons b a)
   a b ... --> (append a (apply append b))))

> (append '(1 2 3) '(a b c) '(x y))
'(1 2 3 a b c x y)
> (append '(a b c))
'(a b c)
> (append '(a b c) 5)

  Signature violation:  expected: list?
 given: 5
  guilty party:  top-level
  innocent party:  function append
  signature:  append : (list? list? .. -> list?)
```

Following function has first argument of functional type:

```
(:: any ((Num -> Bool) (list: Num ..) -> Bool)
  (define/c (any pred)
    (foldr (∘ or pred) #f)))

> (any odd? '(1 2 3 4))
#t
> (any odd? '(0 2 8 4))
#f
> (any + '(0 2 8 4))
```

Signature violation: expected a procedure that accepts 1 mandatory argument without any keywords
given: #<procedure:+>
guilty party: top-level
innocent party: function any
signature: any : ((Num -> Bool) (listof Num) -> Bool)

```
> (any odd? '(0 x 8 4))
```

Signature violation: expected: Num
given: 'x
guilty party: top-level
innocent party: function any
signature: any : ((Num -> Bool) (listof Num) -> Bool)

The signature of function having optional arguments:

```
(:: range (Real (? Real Real) -> (list: Real ..))
  (define range
    (case-lambda
      [(n) (range 1 n)]
      [(a b) (range a b 1)]
      [(a b s) (let loop ([i a] [res '()])
                 (if (< (- b i (- s)) s)
                     (reverse res)
                     (loop (+ i s) (cons i res))))])))

> (range 4)
'(1 2 3 4)
> (range 3 8)
'(3 4 5 6 7 8)
> (range -1 1 1/2)
'(-1 -1/2 0 1/2 1)
```

Polymorphic types

The following signature uses polymorphic types `A` and `B`.

```
(:: bind (A (A -> B) -> B)
  (define (bind x f)
    (f x)))

> (bind 4 sqrt)
2
> (bind 's cons)
```

Signature violation: expected a procedure that accepts 1 mandatory argument without any keywords
given: #<procedure:cons>
guilty party: top-level
innocent party: function bind
signature: bind : (parametric->/c (A B) ...)

Here is the signature of the `foldr` function:

```
(:: foldr ((a b -> b) b (list: a ..) -> b)
  (define/c (foldr f x0)
    (/. '() --> x0
        (cons h t) --> (f h (foldr f x0 t)))))

> (foldr ($ 'f) 'x0 '(1 2 3))
'(f 1 (f 2 (f 3 x0)))
> (foldr + 0 '(1 2 3))
6
```

6 Monads

The bindings documented in this section are provided by the `formica/monads` library and `formica` language.

Monads give very elegant and powerful abstraction. In different forms monads exist in any programming language, for example the `let*` form could be considered as limited representation of the `Identity` monad, and `for*/list` iterator provides core functionality of the List monad. Compare

Racket

```
(let* ([x 5]
       [x (+ x 3)])
  (f x))
```

Haskell

```
do x <- 5
   x <- x + 3
   f x
```

Racket

```
(for*/list ([x '(1 2 3 6 9)]
            [y (in-range x)]
            #:when (even? y))
  (+ x y))
```

Haskell

```
do x <- [1,2,3,6,9]
   y <- [0 .. x]
   guard y % 2 = 0
   return (f + x)
```

The possibility to define and use arbitrary monads allows to separate semantics and syntax, according to the well-known Alan Perlis' principle of "a hundred tools for one data structure". In Racket there are 28 different `for` iterators and creating an iterator with new semantics requires non-trivial syntax engineering (according to examples in the Racket Reference Guide). The syntax tools for monads are limited by binding (`>>=`) and sequential binding (`do` and `collect`) forms, which could be used in uniform way with *any* monad. Moreover, for *any* monad it is possible to use lifting and composition (`lift`, `lift/m`, `compose/m`), folding (`fold/m`, `map/m`, `filter/m`) and guarding (for additive monads). The semantics of all these forms and functions is completely defined only by setting the `return` and `bind` functions (complemented by `mplus` and `mzero` in case additive monads).

Thus, even though the Racket language does not need monads for imperative programming or list comprehension, as Haskell do, monads provide useful and powerful concept for program design.

6.1 Basic operations with monads

6.1.1 Constructing monads

Monads are first-class objects in Formica. Following function returns anonymous monad which could be used to create parameterized monads as mixins.

```
(monad  #:return return
        #:bind bind
       [#:mzero mzero
        #:mplus mplus
        #:type type
        #:failure failure]) → monad?
  return : (Any → Any)
  bind : (Any (Any → Any) → Any)
  mzero : Any = 'undefined
  mplus : (Any Any → Any) = 'undefined
  type : contract? = #f
  failure : (Any → any) = raise-match-error
```

Returns a monad or an additive monad with given *return* and *bind* functions, complemented by *mplus* operation and zero element *mzero* in case of additive monads.

Keywords `#:return` and `#:bind` are used for clarity and can't be omitted, however they could be mixed in arbitrary order with others.

If the *type* contract is given, monadic values are restricted to satisfy the contract. The concept of monads fits the type system very well, providing type consistency in sequential computations. The type specification helps to debug programs using monads and makes them more robust.

If the *failure* function is given, it will be called in case of failure of pattern-matching in `do` and `collect` forms. By default it raises an exception.

```
(define-monad id m-expr)
(define-monad id
  #:return return
  #:bind bind
  [#:mzero mzero]
  [#:mplus mplus]
  [#:type type]
  [#:failure failure])
```

```
m-expr : monad?
return : (Any → Any)
bind : (Any (Any → Any) → Any)
mzero : Any
mplus : (Any Any → Any)
type : contract?
failure : (Any → Any)
```

First form defines named monad as a result of `m-expr`. The second form defines named monad in the same way as `monad` constructor.

Examples:

A simple container monad:

```
(define-formal (m 1) f g)

(define-monad M
  #:return m
  #:bind (/. (m x) f --> (f x)))

> (using-monad M)
```

Basic binding:

```
> (return 'a)
'(m a)
> (bind (m 'a) >>= (lift f))
'(m (f a))
> (bind (m 'a) >>= (lift f) >>= (lift g))
'(m (g (f a)))
> (do [x <-: 'a]
      [y <-: (f x 'b)]
      (return (g y)))
'(m (g (f a b)))
```

Monadic functions:

```
> ((compose/m (lift f) (lift g)) 'a)
'(m (f (g a)))
```

```
> (lift/m f (m 'a) (m 'b))
'(m (f a b))
```

A simple additive monad (equivalent to `Maybe`). In this example the type of monadic values is specified.

```
(define-formal (m 1) (z 0))

(define-monad A/M
  #:type (or/c m? z?)
  #:return (/. (z) --> (z)
               x  --> (m x))
  #:bind (/. (z)   f --> (z)
             (m x) f --> (f x))
  #:mzero (z)
  #:mplus (/. (z) x --> x
              x  _ --> x))

> (using-monad A/M)
```

Basic binding:

```
> (return 'a)
'(m a)
> (return (z))
'(z)
> (bind (m 'a) >>= (lift f))
'(m (f a))
> (bind (z) >>= (lift f) >>= (lift g))
'(z)
> (do [x <-: 'a]
      [y <-: (f x 'b)]
      (return (g y)))
'(m (g (f a b)))
```

Guarding:

```
> (bind (m 2) >>= (guardf even?) >>= (lift g))
'(m (g 2))
> (bind (m 2) >>= (guardf odd?) >>= (lift g))
'(z)
```

```
> (do [x <-: 2]
      (guard (odd? x))
      [y <-: (f x 'b)]
      (return (g y)))
'(z)
> (collect (g y)
    [x <-: 2]
    (odd? x)
    [y <-: (f x 'b)])
'(z)
```

Monadic functions:

```
> ((compose/m (lift f) (lift g)) 'a)
'(m (f (g a)))
> (lift/m f (m 'a) (m 'b))
'(m (f a b))
> (lift/m f (m 'a) (z))
'(z)
> (sum/m '((z) (z) (z) a b))
'a
```

The definition of the `A/M` monad declares the type of monadic values. It makes the error reports more clear.

```
> (bind 'a >>= (lift f))
bind: the argument should have type ((or/c m? z?))
   given: 'a
   using: #<monad:A/M>
> (bind (m 'a) >>= f)
bind: the result should have type ((or/c m? z?))
   received: '(f a)
   using: #<monad:A/M>
> (lift/m f (m 'a) 'b)
bind: the argument should have type ((or/c m? z?))
   given: 'b
   using: #<monad:A/M>
```

Example:

A monad with parameterized type (`Maybe a`):

```
(define-formal Just)
```

```
(define-type (Maybe? a) (Just: a) 'Nothing)

(define (Maybe a)
  (monad
   #:type (Maybe? a)
   #:return (/. 'Nothing --> 'Nothing
                x       --> (Just x))
   #:bind (/. 'Nothing f --> 'Nothing
              (Just x) f --> (f x))
   #:mzero 'Nothing
   #:mplus (/. 'Nothing x --> x
               x       _ --> x)))

> (using-monad (Maybe Int))

> (bind (Just 2) >>= (lift sqr) >>= (lift (* 2)))
'(Just 8)
> (bind 'Nothing >>= (lift sqr) >>= (lift (* 2)))
'Nothing
> (bind (Just 4) >>= (lift sqrt))
'(Just 2)
> (bind (Just 2) >>= (guardf even?) >>= (lift (* 2)))
'(Just 4)
> (bind (Just 2) >>= (guardf odd?) >>= (lift (* 2)))
'Nothing
> (sum/m '(Nothing Nothing (Just 5)))
'(Just 5)
```

Examples with invalid types:

```
> (bind 4 >>= (lift sqrt))
bind: the argument should have type ((Maybe? Int))
  given: 4
  using: #<monad>
> (bind (Just 2) >>= (lift sqrt))
return: the result should have type ((Maybe? Int))
  received: '(Just 1.4142135623730951)
  using: #<monad>
```

We may use any type inside Maybe:

```
> (using (Maybe Sym) (map mplus '(x y z) '(y z x)))
'(x y z)
```

```
(monad? v) → Bool
  v : Any
(monad-zero? v) → Bool
  v : Any
(monad-plus? v) → Bool
  v : Any
```

Return `#t` if `v` is monad, monad with zero element or additive monad, respectively. Otherwise return `#f`.

Examples:

```
> (monad? Id)
#t
> (monad-zero? Id)
#f
> (monad-plus? Id)
#f
> (monad? List)
#t
> (monad-zero? List)
#t
> (monad-plus? List)
#t
```

6.1.2 Switching between monads

All monads share the same syntax for binding and monadic functions. At a given time only one monad, called *currently used monad* could be used.

```
(using-monad) → monad?
(using-monad m) → void?
  m : monad?
```

Defines the currently used monad.

Examples:

```
> (using-monad)
#<monad>
> (using-monad Id)
```

```
> (using-monad)
#<monad:Id>
```

```
(using m expr ...)

  m : monad?
```

Evaluates *expr ...* using monad *m* as currently used monad.

Examples:

```
> (using Id
    (do [x <- 5]
        [x <- (+ x 6)]
        (return x)))
11
> (using List
    (do [x <- '(1 2 3)]
        [x <-: (+ x 6)]
        (return x)))
'(7 8 9)
```

6.1.3 Monadic computations

```
(bind m <arr> f [ <arr> fs ...])

<arr> = >>=
      | >>
```

Binding in the currently used monad.

- *m* >>= *f* binding of function *f* and value *m*.
- *expr1* >> *expr2* sequential computation of *expr1* and *expr2*. Makes sense only if *expr1* has side effects.

The form mimics the *Haskell* syntax for monadic binding:

Haskell: `m >>= f >> g ...`

Formica: `(bind m >>= f >> g ...)`

Examples:

```
> (using Id
    (bind 5 >>= sqr))
25
> (using List
    (bind '(1 2 3) >>= (lift sqr)))
'(1 4 9)
```

```
> (using Id
    (bind 8 >>= displayln >> 5 >>= sqr))
8
25
> (using List
    (bind '(1 2 3) >>= (lift displayln) >> '(4 5 6) >>= (lift sqr)))
1
2
3
'(16 25 36 16 25 36 16 25 36)
```

When called without subform, `bind` evaluates to a binding function of the currently used monad.

```
> (using List
    (apply bind (list '(a b c) (lift f))))
'((f a) (f b) (f c))
```

```
(do ops ...+ res)

ops = (pat <- expr)
    | (pat <-: expr)
    | ((pat1 pat2 ...) <<- expr)
    | ((pat1 pat2 ...) <<-: expr)
    | expr
```

Performs sequential computations in the context of the currently used monad. Mimics do-syntax of *Haskell* language.

Operators *ops* could have any of following forms:

- `(pat <- expr)` matches expression *expr* with pattern *pat* and binds named patterns.

- `(pat <-: expr)` equivalent to `(pat <- (return expr))`.
- `((pat1 pat2 ...) <<- expr)` equivalent to a sequence `(pat1 <- expr) (pat2 <- expr) ...`.
- `((pat1 pat2 ...) <<-: expr)` equivalent to a sequence `(pat1 <-: expr) (pat2 <-: expr) ...`.
- `(expr)` evaluates `expr` for side effects or guarding.

Examples:

```
(define-formal f)
```

Simple monadic binding:

```
> (using List
    (do [x <- '(1 2 3)]
        [y <- '(a b c)]
        (return (f x y))))
'((f 1 a) (f 1 b) (f 1 c) (f 2 a) (f 2 b) (f 2 c) (f 3 a) (f 3 b)
(f 3 c))
```

Using pattern-matching

```
> (using List
    (do [(cons x y) <- '((1 . 2) (3 . 4))]
        (return (f x y))))
'((f 1 2) (f 3 4))
```

Here the `x` is binds to a whole list, not to it's elements:

```
> (using List
    (do [x <-: '(1 2 3)]
        [y <- '(a b c)]
        (return (f x y))))
'((f (1 2 3) a) (f (1 2 3) b) (f (1 2 3) c))
```

Sequential binding:

```
> (using List
    (do [(x y) <<- '(1 2 3)]
        (return (f x y))))
'((f 1 1) (f 1 2) (f 1 3) (f 2 1) (f 2 2) (f 2 3) (f 3 1) (f 3 2)
(f 3 3))
```

```
> (using List
    (do [(x y) <<-: '(1 2 3)]
        [z <- x]
        (return (f x y z))))
'((f (1 2 3) (1 2 3) 1) (f (1 2 3) (1 2 3) 2) (f (1 2 3) (1 2 3)
3))
```

Guarding

```
> (using List
    (do [(x y) <<- '(1 2 3)]
        (guard (< x y))
        (return (f x y))))
'((f 1 2) (f 1 3) (f 2 3))
```

```
(collect expr ops ...+)

ops = (pat <- expr)
    | (pat <-: expr)
    | ((pat1 pat2 ...) <<- expr)
    | ((pat1 pat2 ...) <<-: expr)
    | guard-expr
```

Performs monadic set comprehension. Defined for monads with zero element.

Uses the same binding syntax as `do` form, except for guarding. Any *guard-expr* is evaluated and used as am argument of the `guard` function.

Examples:

```
> (using List
    (collect (cons x y) [x <- '(1 2)] [y <- '(a b c)]))
'((1 . a) (1 . b) (1 . c) (2 . a) (2 . b) (2 . c))
```

Sequential binding:

```
> (using List
    (collect (cons x y) [(x y) <<- '(1 2 3)]))
'((1 . 1) (1 . 2) (1 . 3) (2 . 1) (2 . 2) (2 . 3) (3 . 1) (3 . 2)
(3 . 3))
```

Using guard:

```
> (using List
    (collect (cons x y) [(x y) <<- '(1 2 3)] (< x y)))
'((1 . 2) (1 . 3) (2 . 3))
```

```
> (using List
    (collect `((gcd ,x ,y) = ,z)
      [(x y) <<- (range 8)]
      [z <- (range 2 x)]
      (= z (gcd x y))))
'(((gcd 4 2) = 2)
  ((gcd 4 6) = 2)
  ((gcd 6 2) = 2)
  ((gcd 6 3) = 3)
  ((gcd 6 4) = 2))
```

```
(undefined) → undefined?
```

Represents an object which satisfies following relations in any monad:

$$\texttt{(return (undefined))} \equiv \texttt{(undefined)}$$

$$\texttt{(bind (undefined) >>= } f\texttt{)} \equiv \texttt{(}f\texttt{ (undefined))}$$

The `(undefined)` object could be used as the *unit type* `()` in *Haskell*.

6.1.4 Monadic functions and operators

```
(return x) → any
  x : Any
```

The unit function of currently used monad.

Examples:

```
> (using Id (return 5))
5
> (using List (return 1 2 3))
'(1 2 3)
```

```
mzero : Any
```

A zero element of the currently used monad.

Examples:

```
> (using List mzero)
'()
> (using Id mzero)
mzero: #<monad:Id> is not of a type <monad-zero?>
```

```
(mplus x y) → Any
  x : Any
  y : Any
```

A monadic plus operation of the currently used monad.

Examples:

```
> (using List (mplus '(1 2 3) '(3 4 5)))
'(1 2 3 3 4 5)
```

```
(failure v) → any
  v : Any
```

A failure function which is called if pattern-matching fails in `do` form.

Examples:

```
> (using Id (failure 'x))
do: no matching clause for x
> (using List (failure 'x))
'()
```

```
(lift f) → Fun
  f : Fun
```

Returns a function `f` lifted into the currently used monad.

```
lift f = return ∘ f
```

Examples:

```
> (using List ((lift +) 1 2))
'(3)
```

`(lift/m f arg ...+)`

Monadic application of function `f` to arguments `arg ...`.

```
(lift/m f a b ...) ≡ (do [x <- a]
                         [y <- b]
                         ...
                         [return (f x y ...)])
```

In *Scheme* the more suitable name for this form would be `apply/m`, but here we follow the *Haskell* tradition.

Examples:

```
> (using List
    (lift/m cons '(a b c) '(x y)))
'((a . x) (a . y) (b . x) (b . y) (c . x) (c . y))
```

```
> (using List
    (lift/m or '(#t #f) '(#t #f)))
'(#t #t #t #f)
```

`(compose/m fs ...+)`

Monadic composition of functions `fs`.

```
(compose/m f g ...) ≡
  (λ (x)
    (bind (return x) >>= f >>= g >>= ...))
```

Example: monadic composition in the `List` monad allows to compose functions returning several values:

```
(using-monad List)

(define (Sqrt x)
  (do (guard (positive? x))
      [r <-: (sqrt x)]
      (list r (- r))))

> (Sqrt 4)
'(2 -2)
> ((compose/m Sqrt Sqrt) 16)
'(2 -2)
```

For more examples see "`List-monad.rkt`" file in the "`examples/`" folder.

`(guard test)` → `Any`
  test : `Bool`

Guarding operator. Defined for monads with zero.

```
(guard test) ≡ (if test (return (undefined)) mzero)
```

Examples:

```
> (using List
    (do [(x y) <<- '(1 2 3)]
        (guard (odd? x))
        (guard (< x y))
        (return (cons x y))))
'((1 . 2) (1 . 3))
```

Using `guard` it is possible to perform a backtracking search:

```
> (define tell (lift printf))

> (using List
    (collect (cons x y)
      [x <- '(1 2 3 4)]
      (tell "x: ~a\n" x)
      (odd? x)
```

```
      [y <- '(1 2 3)]
      (tell "x: ~a\ty: ~a\n" x y)
      (< x y)))
x: 1
x: 1 y: 1
x: 1 y: 2
x: 1 y: 3
x: 2
x: 3
x: 3 y: 1
x: 3 y: 2
x: 3 y: 3
x: 4
'((1 . 2) (1 . 3))
```

For more examples see "nondeterministic.rkt" file in the "examples/" folder.

(guardf *pred*) → any
pred : (Any --> Bool)

Guarding function. Defined for monads with zero.

```
(guardf pred) = (bind (guard (pred x)) >> (return x))
```

Examples:

```
> (using List
    (bind '(1 2 3) >>= (guardf odd?) >>= (lift sqr)))
'(1 9)
```

(seq/m *lst*) → Any
lst : list?

Monadic sequencing.

```
(seq/m lst) = (foldr (lift/m cons) (return '()) lst)
```

Examples:

```
> (using List
    (seq/m '((1 2) (3 4 5))))
'((1 3) (1 4) (1 5) (2 3) (2 4) (2 5))
```

```
(map/m f lst) → Any
  f : (Any Any → Any)
  lst : list?
```

Monadic mapping.

```
map/m f = seq/m ∘ (map f)
```

Example (definition of `Sqrt` function see in the example to the `compose/m` operator):

```
> (using List
    (map/m Sqrt '(1 4 9)))
'((1 2 3) (1 2 -3) (1 -2 3) (1 -2 -3) (-1 2 3) (-1 2 -3) (-1 -2 3)
(-1 -2 -3))
```

```
(fold/m f x0 lst) → Any
  f : (Any Any → Any)
  x0 : Any
  lst : list?
```

Monadic fold.

```
(fold/m f x0 '()) = (return x0)
(fold/m f x0 (cons h t)) = (do [y <- (f x0 h)]
                               (fold-m f y t))
```

Examples::

```
> (using List
    (fold/m (lift (hold +)) 0 '(1 2 3)))
'((+ 3 (+ 2 (+ 1 0))))
> (using List
    (fold/m (λ(x y) (list (($ +) x y)
                          (($ -) x y)))
            0
            '(1 2 3)))
'((+ 3 (+ 2 (+ 1 0)))
  (- 3 (+ 2 (+ 1 0)))
  (+ 3 (- 2 (+ 1 0)))
  (- 3 (- 2 (+ 1 0)))
  (+ 3 (+ 2 (- 1 0)))
  (- 3 (+ 2 (- 1 0)))
  (+ 3 (- 2 (- 1 0)))
  (- 3 (- 2 (- 1 0))))
```

```
(filter/m f x0 lst) → Any
  f : (Any Any → Any)
  x0 : Any
  lst : list?
```

Monadic filter.

```
(filter/m pred '()) = (return '())
(filter/m pred (cons h t)) = (do [b <- (pred h)]
                                 [x <- (filter/m pred t)]
                                 (return (if b (cons h x) x)))
```

Examples:

```
> (using List
    (filter/m (lift odd?) '(1 2 3 4 5)))
'((1 3 5))
> (using List
    (filter/m (λ(x) (list #t #f)) '(1 2 3)))
'((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

```
(sum/m lst) → Any
  lst : list?
```

Monadic sum, defined for applicative monads.

```
(sum/m lst) = (foldr mplus mzero lst)
```

Examples:

```
> (using List
    (sum/m '((1 2 3) (2 3 4) '(a b))))
'(1 2 3 2 3 4 quote (a b))
```

6.2 Monads, defined in Formica

6.2.1 The Identity monad

```
Id : monad?
```

The identity monad.

Definition:

```
return = id
bind m f = (f m)
```

Examples:

```
(using-monad Id)
(define-formal f)

> (return 'x)
'x
> (bind 'x >>= f)
'(f x)
```

In the `Id` monad `do` form works like `match-let*` form with ability to produce side effects within calculations:

```
> (do [x <- 5]
      [y <- 8]
      (displayln x)
      [x <- (+ x y)]
      (list x y))
5
'(13 8)


> (do [(cons x y) <- '(1 2 3)]
      [(z t) <<- (reverse y)]
      (return (list x y z t)))
'(1 (2 3) (3 2) (3 2))
```

6.2.2 Ambiguous computations.

The Sequence monad

```
(Sequence  #:return ret
           #:mplus seq-append
          [#:map seq-append-map]) → monad-plus?
  ret : (Any ... → listable?)
```

```
seq-append : (listable? listable? → listable?)
seq-append-map : ((Any → listable?) listable? → listable?)
               = mplus-map
```

Returns a monad which performs computations which may return zero or more then one possible results as a sequence (list, stream, set etc.). This is a generalized monad, monads `List`, `Stream` and `Amb` are instances of the `Sequence` monad.

Definition:

```
return = ret
bind s f = (seq-append-map f s)
mplus = seq-append
mzero = (return)
type = listable?
failure = (const mzero)
```

The sequence of arguments is constructed by the *ret* function. The bound function *f* is applied to all possible values in the input sequence *s* and the resulting sequences are concatenated by the *seq-append* to produce a sequence of all possible results. The details of binding implementation are specified by the mapping function *seq-append-map*.

Examples:

Using `Sequence` it is easy to define monads working with different types of sequences: sets, vectors, strings etc.

```
(define-monad Set
  (Sequence
   #:return set
   #:mplus set-union))

> (using Set
    (lift/m + '(1 2 3) '(2 3 4)))
(set 3 4 5 6 7)
```

```
(define-monad String
 (Sequence
  #:return string
  #:mplus (flipped string-append)))

> (using String
   (collect (char-upcase x)
```

```
      [x <- "abc12xy"]
      (char-alphabetic? x)
      [x <- (format "~a(~a)" x (char->integer x))]))
"A(97)B(98)C(99)X(120)Y(121)"
```

```
(listable? v) → Bool
  v : Any
```

Returns `#t` if `v` is a sequence but not the formal application.

Examples:

```
> (listable? '(1 2 3))
#t
> (listable? 4)
#t
> (listable? (stream 1 2 (/ 0)))
#t
> (listable? '(g x y))
#t
> (define-formal g)

> (listable? (g 'x 'y))
#f
```

```
(mplus-map f s) → listable?
  f : (Any → listable?)
  s : listable?
```

Generalized mapping function for the currently used monad. Formally equal to

```
(mplus-map f s) = (foldl (∘ mplus f) mzero s)
```

```
(zip s ...) → sequence?
  s : listable?
```

Returns a sequence where each element is a list with as many values as the number of supplied sequences; the values, in order, are the values of each sequence. Used to process sequences in parallel, as in `for/list` iterator.

Example of using `zip`:

```
> (using List
    (do [(list x y) <- (zip '(a b c)
                            '(1 2 3 4))]
      (return (f x y))))
'((f a 1) (f b 2) (f c 3))
```

The same with `for/list` iterator.

```
> (for/list ([x '(a b c)]
             [y '(1 2 3 4)])
    (f x y))
'((f a 1) (f b 2) (f c 3))
```

The List monad

```
List : monad-plus?
```

The `List` monad is used for list comprehension and in order to perform calculations with functions, returning more then one value. The main difference between the `List` and monad `[]` in *Haskell*, is the ability of `List` to operate with any sequences as `for` iterators do.

Definition:

```
List = (Sequence #:return list
                 #:mplus concatenate
                 #:map concat-map)
```

```
(concatenate s ...) → list?
  s : listable?
```

Returns a result of *s ...* concatenation in a form of a list.

Examples:

```
> (concatenate '(1 2 3) '(a b c))
'(1 2 3 a b c)
> (concatenate 4 (stream 'a 'b 'c))
'(0 1 2 3 a b c)
> (concatenate (in-set (set 'x 'y 'z)) (in-value 8))
'(x y z 8)
> (concatenate 1 2 3)
'(0 0 1 0 1 2)
```

```
(concat-map f s) → list?
  f : (any/c → n/f-list?)
  s : listable?
```

Applies `f` to elements of `s` and returns the concatenation of results.

Examples:

```
> (concat-map (λ (x) (list x (- x))) '(1 2 3))
'(1 -1 2 -2 3 -3)
> (concat-map (λ (x) (list x (- x))) 4)
'(0 0 1 -1 2 -2 3 -3)
```

Examples

```
(using-monad List)
```

Examples of list comprehension

```
> (collect (sqr x) [x <- '(1 2 5 13 4 24)] (odd? x))
'(1 25 169)
```

```
> (collect (cons x y)
    [x <- '(1 3 4 8 2 4)]
    [y <- '(3 1 6 4 3)]
    (< x y)
    (= (modulo x y) 2))
'((2 . 3) (2 . 6) (2 . 4) (2 . 3))
```

```
> (collect (cons x y)
    [(list x y) <- '((1 2) (2 4) (3 6) (5 1))]
    (odd? x)
    (< x y))
'((1 . 2) (3 . 6))
```

In place of a list any sequence could be used, but only a list is produced.

```
> (collect (cons x y) [(x y) <<- 10] (< 4 (+ (sqr x) (sqr y)) 9))
'((1 . 2) (2 . 1) (2 . 2))
```

```
> (lift/m cons 2 "abc")
'((0 . #\a) (0 . #\b) (0 . #\c) (1 . #\a) (1 . #\b) (1 . #\c))
```

Forms `do` and `collect` in the `List` monad work like `for*/list` form:

```
> (do [x <- 2]
      [y <- "abc"]
      (return (cons x y)))
'((0 . #\a) (0 . #\b) (0 . #\c) (1 . #\a) (1 . #\b) (1 . #\c))
> (for*/list ([x 2]
              [y "abc"])
    (cons x y))
'((0 . #\a) (0 . #\b) (0 . #\c) (1 . #\a) (1 . #\b) (1 . #\c))
```

```
> (collect (cons x y)
    [x <- 3]
    (odd? x)
    [y <- "abc"])
'((1 . #\a) (1 . #\b) (1 . #\c))
> (for*/list ([x 3]
              #:when (odd? x)
              [y "abc"])
    (cons x y))
'((1 . #\a) (1 . #\b) (1 . #\c))
```

It is easy to combine parallel and usual monadic processing:

```
> (using List
    (do [a <- '(x y z)]
        [(list x y) <- (zip "AB"
                            (in-naturals))]
      (return (f a x y))))
'((f x #\A 0) (f x #\B 1) (f y #\A 0) (f y #\B 1) (f z #\A 0) (f z
#\B 1))
```

The use of monad `List` goes beyond the simple list generation. The main purpose of monadic computations is to provide computation with functions which may return more then one value (or fail to produce any). The examples of various applications of this monad could be found in the `"nondeterministic.rkt"` file in the `"examples/"` folder.

The Stream monad

`Stream : monad-plus?`

Like `List` monad, but provides lazy list processing. This monad is equivalent to monad `[]` in *Haskell* and could be used for operating with potentially infinite sequences.

Definition:

```
Stream = (Sequence #:return list
                   #:mplus stream-concatenate
                   #:map stream-concat-map)
```

```
(stream-concatenate s ...) → stream?
  s : listable?
```

Returns a result of `s ...` lazy concatenation in a form of a stream.

Examples:

```
> (stream->list
   (stream-concatenate '(1 2 3) '(a b c)))
'(1 2 3 a b c)
> (stream->list
   (stream-concatenate 4 (stream 'a 'b 'c)))
'(0 1 2 3 a b c)
> (stream-ref
   (stream-concatenate (stream 1 (/ 0)) (in-naturals))
   0)
1
> (stream-ref
   (stream-concatenate (stream 1 (/ 0)) (in-naturals))
   1)
/: division by zero
```

```
(stream-concat-map f s) → stream?
  f : (any/c → stream?)
  s : listable?
```

Applies `f` to elements of `s` and lazily returns the concatenation of results.

Examples:

```
> (stream->list
   (stream-concat-map (λ (x) (stream x (- x))) '(1 2 3)))
'(1 -1 2 -2 3 -3)
```

```
> (stream->list
   (stream-concat-map (λ (x) (stream x (- x))) 4))
'(0 0 1 -1 2 -2 3 -3)
> (stream-ref
   (stream-concat-map (λ (x) (stream x (/ x))) '(1 0 3))
   0)
1
> (stream-ref
   (stream-concat-map (λ (x) (stream x (/ x))) '(1 0 3))
   1)
1
> (stream-ref
   (stream-concat-map (λ (x) (stream x (/ x))) '(1 0 3))
   2)
0
> (stream-ref
   (stream-concat-map (λ (x) (stream x (/ x))) '(1 0 3))
   3)
/: division by zero
```

```
(stream-take s n) → list?
  s : stream?
  n : Nat
```

Returns list of `n` first elements of stream (or sequence) `s`.

Examples:

```
> (stream-take (stream 'a 'b 'c) 2)
'(a b)
> (stream-take (stream 'a 'b (/ 0)) 2)
'(a b)
> (stream-take (in-naturals) 3)
'(0 1 2)
```

```
(scons h t)
```

A match expander which matches non-empty streams and binds the first element to `h`, and the rest of stream to `t`. Only the first element is evaluated eagerly.

Examples:

```
> (require racket/match)
```

```
> (match (in-naturals)
    [(scons h t) (list h t)])
'(0 #<stream>)
> (match (stream 1 (/ 0))
    [(scons h t) (list h t)])
'(1 #<stream>)
```

Examples

```
(using-monad Stream)
```

Two classical examples with infinite sequences.

The infinite sequence of Pythagorean triples:

```
(define triples
 (collect (list a b c)
   [a <- (in-naturals)]
   [b <- (in-range (ceiling (/ a 2)) a)]
   [c <-: (sqrt (- (sqr a) (sqr b)))]
   (integer? c)
   (> b c)))

> (stream-take triples 3)
'((5 4 3) (10 8 6) (13 12 5))
> (stream-ref triples 100)
'(170 150 80)
```

The first triangle with area exceeding 100:

```
> (stream-first
   (collect t
     [(and t (list _ b c)) <- triples]
     (> (* 1/2 b c) 100)))
'(25 20 15)
```

The infinite sequence of primes:

```
(define (primes r)
```

```
  (do [(scons x xs) <-: r]
      [p <-: (collect y
                [y <- xs]
                (not (zero? (modulo y x))))]
      (stream-cons x (primes p))))

> (stream-take (primes (in-naturals 2)) 10)
'(2 3 5 7 11 13 17 19 23 29)
> (stream-ref (primes (in-naturals 2)) 100)
547
```

Using monad `Stream` all monadic functions work lazily however they still operate on eager lists:

```
> (stream-first ((compose/m (lift /) (lift (curry * 2))) 1/2 0))
1
> (stream-first (lift/m / (return 1) (return 1 0)))
1
> (stream-ref (lift/m / (return 1) (return 1 0)) 1)
/: division by zero
> (stream-ref (map/m (λ (x) (stream x (/ x))) '(1 0 3)) 0)
'(1 0 3)
> (stream-ref (map/m (λ (x) (stream x (/ x))) '(1 0 3)) 1)
'(1 0 1/3)
> (stream-ref (map/m (λ (x) (stream x (/ x))) '(1 0 3)) 2)
/: division by zero
```

The Amb monad

```
Amb : monad-plus?
```

Like `Stream` monad, but tries to return a list of unique elements.

Definition:

```
Amb = (Sequence #:return amb
                #:mplus amb-union
                #:map amb-union-map)
```

```
(amb v ...) → stream?
  v : any/c
```

Returns a stream of arguments `v` removing duplicates (in terms of `equal?`).

Examples:

```
> (stream->list (amb 1 3 2 2 3 2 1 2 4 3))
'(1 3 2 4)
> (stream-take (amb 1 2 (/ 0)) 2)
'(1 2)
```

```
(amb-union s1 s2) → stream?
  s1 : listable?
  s2 : listable?
```

Returns a stream of elements from *s1* and *s2*, removing duplicates.

Examples:

```
> (stream->list
    (amb-union '(1 2 3) '(2 3 4)))
'(1 2 3 4)
> (stream->list
    (amb-union 4 (amb 'a 'b 'c)))
'(0 1 2 3 a b c)
> (stream-ref
    (amb-union (amb 1 (/ 0)) (in-naturals))
    0)
1
> (stream-ref
    (amb-union (amb 1 (/ 0)) (in-naturals))
    1)
/: division by zero
```

```
(amb-union-map f s) → stream?
  f : (any/c → stream?)
  s : listable?
```

Applies *f* to elements of *s* and lazily returns the union of results.

Examples:

```
> (stream->list
    (amb-union-map (lift sqr) '(-3 -2 -1 0 -1 2 3)))
'(9 4 1 0)
> (stream->list
    (amb-union-map (λ (x) (amb x (- x))) 4))
```

```
'(0 1 -1 2 -2 3 -3)
> (stream-ref
   (amb-union-map (λ (x) (amb x (/ x))) '(1 0 3))
   0)
1
> (stream-ref
   (amb-union-map (λ (x) (amb x (/ x))) '(1 0 3))
   1)
0
> (stream-ref
   (amb-union-map (λ (x) (amb x (/ x))) '(1 0 3))
   2)
/: division by zero
```

Examples

```
(using-monad Amb)
```

Proving logical statements by brute force

```
> (stream->list
   (collect (eq? (==> A B) (or B (not A)))
     [(A B) <<- (amb #t #f)]))
'(#t)
> (stream->list
   (collect (==> (==> (==> A B) C) (==> A (==> B C)))
     [(A B C) <<- (amb #t #f)]))
'(#t)
> (stream->list
   (collect (eq? (==> (==> A B) C) (==> A (==> B C)))
     [(A B C) <<- (amb #t #f)]))
'(#t #f)
> (stream->list
   (collect (list A B C)
     [(A B C) <<- (amb #t #f)]
     (not (eq? (==> (==> A B) C) (==> A (==> B C))))))
'((#f #t #f) (#f #f #f))
```

7 Miscellaneous issues

7.1 Differences from Racket

In order to use simplified syntax for partial application and curring, the arity of some variadic functions, defined in the Racket is changed in Formica.

```
(+ x y ...+) → Num
  x : Num
  y : Num
(* x y ...+) → Num
  x : Num
  y : Num
```

Same as `+` and `*` defined in `racket/base`, but accept at least two arguments.

```
> (+ 1 2 3)
6
> (+ 1 2)
3
> (+ 1)
#<procedure:curried:+>
> (+)
#<procedure:curried:+>
> (map (+ 1) '(1 2 3))
'(2 3 4)
> (* 1)
#<procedure:curried:*>
> (map (* 2) '(1 2 3))
'(2 4 6)
```

```
(map f lsts ...+) → list?
  f : Fun
  lsts : list?
```

Same as `map`, defined in `racket/base`, but accepts at least one list.

```
> (map cons '(1 2 3) '(a b c))
'((1 . a) (2 . b) (3 . c))
> (map (* 2) '(1 2 3))
'(2 4 6)
> (map (* 2))
#<procedure:curried:map>
> (map (map (* 2)) '((1 2 3) (2 3 4) (5)))
'((2 4 6) (4 6 8) (10))
```

```
(all? f lsts ...+) → Bool
  f : Fun
  lsts : list?
(any? f lsts ...+) → Bool
  f : Fun
  lsts : list?
```

Same as andmap and ormap defined in racket/base, but accept at least one list.

```
> (any? odd? '(1 2 3))
#t
> (all? < '(1 2 3) '(2 3 4))
#t
```

Some forms could be used as functions

```
(or expr ...)
```

```
(and expr ...)
```

Same as or and and forms defined in racket/base, however, evaluate to binary boolean functions if used not in a head position of application form.

```
> or
#<procedure:or>
> and
#<procedure:and>
> (or 1 (/ 1 0))
1
> (and #f (/ 1 0))
#f
> (map or '(#t #f #t #f) '(#t #t #f #f))
'(#t #t #t #f)
> (define/c (any p) (foldr (o or p) #f))

> (define/c (all p) (foldr (o and p) #f))

> (any odd? '(2 4 6 5 8 10))
#t
> (all even? '(2 4 6 5 8 10))
#f
```

```
(==> expr1 expr2)
```

Represents implication. Equivalent to

```
(if expr1 expr2 #t).
```

Returns binary boolean function if used not in a head position of application form.

```
> ==>
#<procedure:|==>|>
> (==> 'A 'B)
'B
> (==> #f (/ 1 0))
#t
```

```
(eq? x y ...+) → boolean?
  x : Any
  y : Any
(equal? x y ...+) → boolean?
  x : Any
  y : Any
```

Same as `eq?` and `equal?` provided `racket/base`, but if more then two arguments are given, arguments are compared pairwise from left to right.

```
(eval expr [ns]) → any
  expr : Any
  ns : namespace? = formica-namespace
```

Evaluates the expression *expr* like `eval` provided `racket/base`, but uses predefined namespace containing bindings provided by the *Formica* language.

7.2 Comparison and ordering

The bindings documented in this section are provided by the `formica/tools` library and `formica` language.

```
(different? v1 v2 ...+) → boolean?
  v1 : Any
  v2 : Any
```

Returns `#t` if *v1* and *v2* are not `equal?`, and `#f` otherwise. If more then two arguments are given, they are compared pairwise, so the result is `#t` if none of arguments are `equal?`.

Examples:

```
> (different? 1 1)
#f
> (different? 1 2)
#t
> (different? 1 1.0)
#t
> (different? 1 2 6 3 9 7)
#t
> (different? 1 2 6 3 2 7)
#f
```

(almost-equal? *v1* *v2* ...+) → boolean?
  v1 : Any
  v2 : Any

≈ : almost-equal?

Returns `#t` if *v1* and *v2* are `equal?`, or for numeric values if the magnitude of difference between *x* and *y* is less then `(tolerance)`. Returns `#f` otherwise.

If lists and pairs are compared, they are `almost-equal?` if they have the same structure and all entries at corresponding positions are `almost-equal?`.

If more then two arguments are given, they are compared pairwise from left to right.

Function `almost-equal?` has an alias: ≈ (could be entered as `\approx` + Alt `\`).

Examples:

```
> (≈ 'x 'x)
#t
> (≈ 1 'x)
#f
> (≈ 1 1.0)
#t
> (≈ 1/2 (+ 0.5 1e-15))
#f
> (≈ 1/2 (+ 0.5 1e-16))
#t
> (≈ 1/2 (+ 0.5 0+1e-16i))
#t
> (≈ 0.5 1/2 (+ 0.5 0+1e-16i))
#t
> (≈ '(1 (2 3)) '(1 (2.000000000000001 3)))
#t
```

```
(tolerance) → real?
(tolerance x) → void?
  x : real?
```

Defines an relative tolerance used by the `almost-equal?` predicate. The default value is `5e-16`.

Examples:

```
> (parameterize ([tolerance 0.01])
    (and (≈ 1 1.001)
         (≈ 10 10.01)
         (≈ 1e+23 1.001e+23)
         (≈ 1e-23 1.001e-23)
         (≈ 0 0.001)))
#t
```

In the last case it is impossible to use the relative tolerance, so `(tolerance)` is interpreted as absolute.

7.2.1 Generic ordering

Symbolic transformations often require ordering of objects of different kinds: numbers, booleans, strings or lists. For example, these two algebraic expressions, should be equivalent:

```
'(+ a b) = '(+ b a).
```

As another example consider following simplification chain:

```
'(+ x 2 (* y x) 5) ==>

'(+ 2 5 x (* x y)) ==>

  '(+ 7 x (* x y)).
```

So it is reasonable to sort the arguments of commutative operations. One more example, is given in the `"logics.rkt"` file in the `"formica/examples"` folder.

Formica provides a generic ordering procedure which allows to define the ordering on different sets which are represented by contract-based types.

By default following ordering of different types is used:

Order	Type	Ordering function
1	`#t`	`(const #f)`
2	`#f`	`(const #f)`
3	`Real`	`<`
4	`Str`	`string<?`
5	`Sym`	`symbol<?`
6	`null?`	`(const #f)`
7	`pair?`	`pair<?`

It means that any string follows any real number, and for comparing values within the type the corresponding ordering function is used. This table is stored in the `(type-ordering)` parameter, and could be extended or modified.

```
(ordered? v ...) → boolean?
  v : Any
```

Returns `#t` if arguments `v ...` are ordered according to ordering given by the `(type-ordering)` parameter. Returns `#f` otherwise.

Examples:

```
> (ordered? 1)
#t
> (ordered? 'x 'y)
#t
> (ordered? 1 "a" 'x)
#t
> (ordered? 1 "a" 'x #f)
#f
> (sort '(#f 'y (3 2) 2.0 "ab" 'x 'b 3 "abc" 'a "bca" (1 (2 3)) (1 2) 1 (1) #t) ordered?)
'(#t #f 1 2.0 3 "ab" "abc" "bca" (1) (1 2) (1 (2 3)) (3 2) 'a 'b
'x 'y)
```

```
(type-ordering) → (list: (cons: contract? (Any Any -> Bool)))
(type-ordering p) → void?
  p : (list: (cons: contract? (Any Any -> Bool)))
```

A parameter which defines the ordering of different types and ordering functions within types.

In following example even numbers follow odd numbers, moreover, within odd numbers reverse ordering is set up. Symbols are not ordered, hence they are left unsorted, but shifted to the right.

```
> (parameterize ([type-ordering (list (cons odd? >)
                                      (cons even? <))])
    (sort '(0 1 2 3 'x 4 5 6 'a 7 8 9) ordered?))
'(9 7 5 3 1 0 2 4 6 8 'x 'a)
```

```
(add-to-type-ordering type
                      [prec-type
                       ord-fun]) → void?
  type : contract?
  prec-type : contract? = 'last
  ord-fun : (Any Any -> Bool) = (cons #f)
```

Adds *type* to the current (type-ordering) table. If the *prec-type* is given, the new type will have the order next to it. If the comparing function *ord-fun* is given, it will be used to compare values within the type.

In this example complex numbers follow reals and precede strings. Moreover, within complex numbers ordering according to magnitude is set up. Symbols are ordered in default way.

```
> (parameterize ([type-ordering (type-ordering)])
    (add-to-type-ordering complex? real? (fork < magnitude))
    (sort '(0-1i 2 3.5 1+2i "x" 4-1i 5 6 "a" 7 -1.0+2.5i 9) ordered?))
'(2 3.5 5 6 7 9 0-1i 1+2i -1.0+2.5i 4-1i "a" "x")
```

```
(symbol<? s1 s2) → boolean?
  s1 : Sym
  s2 : Sym
```

Returns #t if *s1* precedes *s2* in lexicographic order, and #f otherwise.

Examples:

```
> (symbol<? 'x 'y)
#t
> (symbol<? 'abcde 'acde)
#t
> (symbol<? 'x 'x)
#f
> (sort '(a X abc x abcd A) symbol<?)
'(A X a abc abcd x)
```

```
(pair<? p1 p2) → boolean?
  p1 : pair?
  p2 : pair?
```

Defines lexicographic order on a set of pairs, according to ordering of pair elements.

Examples:

```
> (pair<? '(1 2) '(1 2 3))
#t
> (pair<? '(1 2) '(1 3))
#t
> (pair<? '(1 . 2) '(1 2))
#t
> (pair<? '(1 2) '(1 x y))
#t
```

7.3 Operators and functionals

The bindings documented in this section are provided by the `formica/tools` library and `formica` language.

```
(function? x) → Bool
  x : Any
```

Returns `#t` only if `x` is a function and `#f` otherwise.

```
(id x) → Any
  x : Any
```

The identity function.

Examples:

```
> (id 1)
1
> (id 'x)
'x
> (id +)
#<procedure:+>
> ((id +) 1 2)
```

```
3
> (id id)
#<procedure:id>
> (id '(a b c))
'(a b c)
```

```
(arg n) → Fun
  n : Ind
```

Creates a trivial function which returns the *n*-th argument.

Examples:

```
> (arg 1)
#<procedure:arg:1>
> ((arg 1) 'x 'y 'z)
'x
> ((arg 2) 'x 'y 'z)
'y
> ((arg 3) 'x 'y 'z)
'z
```

```
I1 : (arg 1)
I2 : (arg 2)
I3 : (arg 3)
```

Aliases for frequently used `arg` calls.

```
(const x) → Fun
  x : Any
```

Creates a trivial function which returns *x* for any arguments.

Examples:

```
> (const 'A)
#<procedure:const>
> ((const 'A) 1)
'A
> ((const 'A) 1 2)
'A
> ((const +))
#<procedure:+>
```

```
(composition f ...) → Fun
  f : Fun
```

```
∘ : composition
```

Returns a generalized composition of functions `f ...`. Function may have any arity: composition is done as if they all were curried.

In the generalized composition variadic or polyadic functions have minimal possible arity, unless they are greedy.

Function `composition` has an alias: ∘ (could be entered as `\circ` + Alt `\`).

Examples:

```
> (define-formal (u 1) (b 2) (t 3))

> ((∘ u b) 1 2)
'(u (b 1 2))
> ((∘ b u) 1 2)
'(b (u 1) 2)
> ((∘ t b u) 1 2 3 4)
'(t (b (u 1) 2) 3 4)
> ((∘ t u b) 1 2 3 4)
'(t (u (b 1 2)) 3 4)
> ((∘ u t b) 1 2 3 4)
'(u (t (b 1 2) 3 4))
> ((∘ u b t) 1 2 3 4)
'(u (b (t 1 2 3) 4))
> ((∘ b u t) 1 2 3 4)
'(b (u (t 1 2 3)) 4)
> ((∘ b t u) 1 2 3 4)
'(b (t (u 1) 2 3) 4)
> ((∘ length (filter odd?)) '(1 2 3 4))
2
```

Composition with nullary function:

```
> (define-formal (n 0) (u 1) (b 2))

> ((∘ u n))
'(u (n))
```

```
> ((o b n) 1)
'(b (n) 1)
> ((o n u) 1)
nest: Cannot nest function with zero arity!
  function: #<procedure:formal:n>
  position: 2
```

Composition is associative:

```
> ((o (o b t) u) 1 2 3 4)
'(b (t (u 1) 2 3) 4)
> ((o b (o t u)) 1 2 3 4)
'(b (t (u 1) 2 3) 4)
```

Composition has left and right neutral element:

```
> ((o b id) 1 2)
'(b 1 2)
> ((o id b) 1 2)
'(b 1 2)
```

In the generalized composition variadic or polyadic functions have minimal possible arity:

```
> (define (f . x) (cons 'f x))

> (define (g x . y) (list* 'g x y))

> ((o f g) 1 2 3 4)
'(f (g 1) 2 3 4)
```

(greedy *f*) → Fun
  f : Fun

For variadic or polyadic function *f* returns equivalent function having maximal possible arity.

Examples:

```
> (define-formal f g h)
```

```
> ((∘ f g h) 1 2 3 4)
'(f (g (h 1)) 2 3 4)
> ((∘ f (greedy g) h) 1 2 3 4)
'(f (g (h 1) 2 3 4))
> ((∘ remove-duplicates (greedy append)) '(1 2 3) '(2 3 2 4))
'(1 2 3 4)
```

```
(negated p) → Fun
  p : Fun
(¬ p) → Fun
  p : Fun
```

Returns the negation of a predicate *p*.

For function `negated` there is an alias: `¬` (could be entered as `\neg` + Alt `\`).

Examples:

```
> (negated odd?)
#<procedure:negated:odd?>
> ((negated odd?) 2)
#t
> ((¬ <) 1 2)
#f
```

```
(flipped f) → Fun
  f : Fun
```

Returns a function which is functionally equivalent to *f*, but gets it's arguments in reversed order.

Examples:

```
> (define snoc (flipped cons))

> snoc
#<procedure:flipped:cons>
> (snoc 1 2)
'(2 . 1)
> ((flipped list) 1 2 3)
'(3 2 1)
```

```
(fif p f g) → Fun
  p : Fun
  f : Fun
  g : Fun
```

Returns function

```
x y ... = (if (p x y ...) (f x y ...) (g x y ...)).
```

Examples:

```
> (map (fif odd? (+ 1) id) '(0 1 2 3 4))
'(0 2 2 4 4)
```

```
(andf f g ...) → Fun
  f : Fun
  g : Fun
```

Returns function

```
x y ... = (and (f x y ...) (g x y ...) ...).
```

Examples:

```
> (map (andf integer? positive?) '(-3/5 -1 0 2 4.2))
'(#f #f #f #t #f)
```

```
(orf f g ...) → Fun
  f : Fun
  g : Fun
```

Returns function

```
x y ... = (or (f x y ...) (g x y ...) ...).
```

Examples:

```
> (map (orf integer? positive?) '(-3/5 -1 2 4.2))
'(#f #t #t #t)
```

```
(fork f g) → Fun
  f : Fun
  g : unary?
(-< f g) → Fun
  f : Fun
  g : unary?
```

Returns function

```
x y ... = (f (g x) (g y) ...).
```

This function has an alias `-<`.

Examples:

```
> ((fork cons sqr) 2 3)
'(4 . 9)
> ((-< + sqr) 1 2 3)
14
```

```
(all-args p) → Fun
  p : unary?
```

Returns

```
(fork and p).
```

Examples:

```
> ((all-args real?) 2 -3 4.5)
#t
> ((all-args real?) 2 'x 4.5)
#f
```

```
(any-args f) → Fun
  f : unary?
```

Returns function

```
(fork or p).
```

Examples:

```
> ((any-args real?) '(1 2) 'a "abc" 1-2i)
#f
> ((any-args real?) 'x 2 "abc" 0+8i)
#t
```

```
(curry f arg ...) → (or/c curried? Any)
  f : Fun
  arg : Any
(curried f arg ...) → (or/c curried? Any)
  f : Fun
  arg : Any
```

Return partially applied (curried) function *f*, with fixed arguments *arg ...*. Symbols curry and curried are synonyms.

```
(curryr f arg ...) → (or/c curried? Any)
  f : Fun
  arg : Any
(r-curried f arg ...) → (or/c curried? Any)
  f : Fun
  arg : Any
```

Like curry but arguments *arg ...* are fixed from the right side of argument sequence. Symbols curryr and r-curried are synonyms.

Examples of partial application:

```
> (curry list 1 2)
#<procedure:curried:list>
> ((curry list 1 2) 3 4)
'(1 2 3 4)
> (map (curried cons 1) '(1 2 3))
'((1 . 1) (1 . 2) (1 . 3))
> (curryr list 1 2)
#<procedure:curried:list>
> ((curryr list 1 2) 3 4)
'(3 4 1 2)
> (map (r-curried cons 1) '(1 2 3))
'((1 . 1) (2 . 1) (3 . 1))
> (curry cons 1 2)
'(1 . 2)
> (curryr cons 1 2)
'(1 . 2)
```

The curry (curried) and curryr (r-curried) functions correctly reduce the arity of curried function:

```
> (procedure-arity cons)
2
> (procedure-arity (curried cons))
2
> (procedure-arity (curry cons 1))
1
> (procedure-arity (curryr cons 1))
1
> (procedure-arity (curry + 1 2 3))
(arity-at-least 0)
```

```
(curried? x) → boolean?
  x : Any
```

Returns #t if x is partially applied or curried function, and #f otherwise.

Examples:

```
> (curried? (curry cons 1))
#t
> (curried (curry +))
#<procedure:curried:+>
> (curried (curryr +))
#<procedure:curried:+>
> (curried? +)
#f
```

```
(fixed-point f [#:same-test same?]) → Fun
  f : Fun
  same? : (any/c any/c -> boolean?) = equal?
```

Returns a function

```
x = (f (f (f ... (f x))))
```

which finds a least fixed point of *f* by iterative application, while result keeps changing in the sense of the *same?* function.

If function *f* is not unary, it must return as many values, as it could accept.

Example:

```
> (define fcos (fixed-point cos))

> (fcos 1)
0.7390851332151607
> (cos (fcos 1))
0.7390851332151607
```

Let's define naive implementations of secant, bisection and Newton's methods for numerical solution of equation $F(x) = 0$.

```
(define (bisection f)
  (λ (a b)
    (let ([c (* 0.5 (+ a b))])
      (if (<= (* (f a) (f c)) 0)
          (values a c)
          (values c b)))))

(define (secant f)
  (λ (x y)
    (let ([fx (f x)]
          [fy (f y)])
      (values y (/ (- (* x fy) (* y fx))
                   (- fy fx))))))

(define (newton f)
 (λ (x)
   (let* ([ϵ (tolerance)]
          [fx (f x)]
          [dfx (/ (- (f (+ x ϵ)) fx) ϵ)])
     (- x (/ fx dfx)))))
```

Note that functions `bisection` and `secant` require and produce two approximation points on each step, while `newton` function accepts only one point.

Now we can define the universal function `find-root` which solves given equation using these methods:

```
(define (find-root f #:method (method secant))
  (fixed-point (method f) #:same-test almost-equal?))

> (parameterize ([tolerance 1e-10])
    ((find-root (λ (x)(- (cos x) x))
```

```
                #:method secant) 0.0 1.0))
0.7390851332151607
0.7390851332151607
> (parameterize ([tolerance 1e-10])
    ((find-root (λ (x)(- (cos x) x))
                #:method bisection) 0.0 1.0))
0.7390851331874728
0.7390851333038881
> (parameterize ([tolerance 1e-10])
    ((find-root (λ (x)(- (cos x) x))
                #:method newton) 1.0))
0.7390851332151607
```

The `find-root` function inherits arity of it's methods:

```
> (procedure-arity
   (find-root (λ (x)(- (cos x) x))
              #:method secant))
2
> (procedure-arity
   (find-root (λ (x)(- (cos x) x))
              #:method bisection))
2
> (procedure-arity
   (find-root (λ (x)(- (cos x) x))
                 #:method newton))
1
```

7.4 Memoization

The bindings documented in this section are provided by the `formica/memoize` library and `formica` language.

```
(memoized f) → memoized?
  f : Fun
```

Returns memoized equivalent of the function *f*.

Examples:

```
> (define g (memoized current-inexact-milliseconds))
```

```
> (g)
1358217850330.706
> (current-inexact-milliseconds)
1358217850332.17
> (g)
1358217850330.706
```

```
(memoized? f) → Bool
  f : Fun
```

Returns `#t` if *f* is a memoized function, and `#f` otherwise.

Examples:

```
> (define g (memoized +))

> (memoized? g)
#t
> (memoized? +)
#f
> (g 2 3)
5
```

```
(define/memo (f var ...) body ...)
(define/memo f fun)
```

Defines memoized function *f*. If formal variables are not given, the right-hand side of definition *fun* should evaluate to a function.

Examples:

```
> (define/memo (f x)
    (display x)
    (* x x))

> (memoized? f)
#t
> (f 2)
2
4
> (f 2)
4
> (f 3)
3
9
```

```
(define/memo/c (f var ...) body)
```

Defines memoized function `f` using point-free notation.

7.5 Tagged functions

The bindings documented in this section are provided by the `formica/tools` library and `formica` language.

Tagging allows to distinguish functions and classes of functions.

```
(tagged? f) → tagged?
  f : Fun
```

Returns `#t` if `f` is tagged function, and `#f` otherwise.

```
(tag f) → Sym
  f : tagged?
```

Returns a tag of a tagged function `f`.

```
(check-tag f t) → Bool
  f : tagged?
  t : Any
```

Returns `#t` if `f` is a tagged function and has tag `t`, otherwise returns `#f`.

```
(set-tag t [n]) → (Fun -> tagged?)
  t : Sym
  n : (or/c symbol? #f) = #f
```

Returns an operator, which sets a tag `t` to given function. If name `n` is given, tagged function will be renamed.

```
(set-tag* t [n]) → (Fun -> Fun)
  t : Sym
  n : (or/c symbol? #f) = #f
```

Returns an operator, which adds symbol `t` to function name. The result is not a tagged function. If name `n` is given, the function will be renamed.

Examples:

```
> (define t-cons ((set-tag 'tag) cons))

> t-cons
#<procedure:tag:cons>
> (tag t-cons)
'tag
> (check-tag 'tag t-cons)
#t
> (check-tag 'tag cons)
#f
> (tagged? t-cons)
#t


> (define r-cons ((set-tag* 'tag) cons))

> r-cons
#<procedure:tag:cons>
> (check-tag 'tag r-cons)
#f
> (tagged? r-cons)
#f
```

7.6 Managing the function arity

The bindings documented in this section are provided by the `formica/tools` library and `formica` language.

In the reference guide to the Formica language following terminology concerned to function arity is used:

- Function has *fixed arity* if it may accept exact finite number of arguments. The arity of function having fixed arity is expressed as positive integer number.
- Function is called *variadic* if it may accept different (probably unlimited) number of arguments. The arity of variadic function is expressed with the `arity-at-least` structure.
- Function is called *polyadic* if it may accept different but limited number of arguments. The arity of polyadic function is expressed as a list of positive integers or `arity-at-least` structure.

```
(fixed-arity? v) → boolean?
  v : Any
```

Returns `#t` if *v* is a function having fixed arity, and `#f` otherwise.

Examples:

```
> (fixed-arity? (lambda (x) (+ 2 x)))
#t
> (define (f . x) (cons 'f x))

> (fixed-arity? f)
#f
> (fixed-arity? cons)
#t
> (fixed-arity? +)
#f
```

```
(variadic? v) → boolean?
  v : Any
```

Returns `#t` if *v* is variadic function, and `#f` otherwise.

Examples:

```
> (variadic? (case-lambda
               [(x) x]
               [(x . y) (apply + x y)]))
#t
> (define (f . x) (cons 'f x))

> (variadic? f)
#t
> (variadic? cons)
#f
> (variadic? +)
#t
```

```
(polyadic? v) → boolean?
  v : Any
```

Returns `#t` if *v* is polyadic function, and `#f` otherwise.

Examples:

```
> (polyadic? (case-lambda
```

```
                [(x) x]
                [(x y) (+ x y)]))
#t
> (define (f x (y 2)) '(f x y))

> (polyadic? f)
#t
> (polyadic? cons)
#f
> (polyadic? +)
#f
```

(nullary? v) → boolean?
  v : Any

Returns #t if v is a function and arity of v includes 0, and #f otherwise.

Examples:

```
> (nullary? (case-lambda
                [() 0]
                [(x) x]
                [(x . y) (apply + x y)]))
#t
> (define (f . x) (cons 'f x))

> (nullary? f)
#t
> (nullary? cons)
#f
> (nullary? +)
#f
```

(unary? v) → boolean?
  v : Any

Returns #t if v is a function and arity of v includes 1, and #f otherwise.

Examples:

```
> (unary? (case-lambda
                [() 0]
                [(x) x]
```

```
              [(x . y) (apply + x y)]))
#t
> (define (f x . y) (list* 'f x y))

> (unary? f)
#t
> (unary? not)
#t
> (unary? cons)
#f
> (unary? +)
#f
```

(binary? *f*) → boolean?
  f : Any

Returns `#t` if *v* is a function and arity of *v* includes 2, and `#f` otherwise.

Examples:

```
> (binary? (case-lambda
              [() 0]
              [(x y) (+ x y)]
              [(x . y) (apply + x y)]))
#t
> (define (f x . y) (list* 'f x y))

> (binary? f)
#t
> (binary? not)
#f
> (binary? cons)
#t
> (binary? +)
#t
```

(min-arity *f*) → (or/c 0 positive?)
  f : Fun

Returns the minimal arity of a function `#t`.

Examples:

```
> (min-arity (case-lambda
              [() 0]
              [(x y) (+ x y)]))
0
> (define (f x . y) (list* 'f x y))

> (min-arity f)
1
> (min-arity not)
1
> (min-arity cons)
2
> (min-arity +)
2
```

(max-arity *f*) → (or/c 0 positive? +inf.0)
  f : Fun

Returns the maximal arity of a function `#t`.

Examples:

```
> (max-arity (case-lambda
              [() 0]
              [(x y) (+ x y)]))
2
> (define (f x . y) (list* 'f x y))

> (max-arity f)
+inf.0
> (max-arity not)
1
> (max-arity cons)
2
> (max-arity +)
+inf.0
```

Bibliography

[Baader99] Baader, F. and Nipkow, T., *Term Rewriting and All That*, Cambridge University Press, 1999. http://books.google.com/books?id=N7BvXVUCQk8C

[Bezem03] Bezem, M., Klop, J.W. and Vrijer, R., *Term rewriting systems*, Cambridge tracts in theoretical computer science. Cambridge University Press, 2003. http://books.google.com/books?id=oe3QKzhFEBAC

[Turchin89] Turchin, V. F., *REFAL-5 Programming Guide and Reference Manual*, The City College of New York, New England Publishing Co., Holyoke., 1989.

[Surhone10] Surhone, L.M., Timpledon, M.T. and Marseken, S.F., *Refal*, VDM Verlag Dr. Mueller AG & Co. Kg, 2010. http://books.google.com/books?id=SH2rcQAACAAJ

[Wolfram] "Wolfram Mathematica." http://www.wolfram.com/mathematica/