Egison

Release 4.0.0

Jul 13, 2020

Contents

CHAPTER 1

Tutorial

1.1 Getting Started

1.1.1 Installation

See *Installation* for the install guide of the Egison interpreter.

1.1.2 How to use the Egison interpreter

Interactive mode

Just typing `egison` starts the REPL (read-eval-print loop) of the interpreter.

```
$ egison
```

You can load definitions from a file by passing `-l` option. The following example is equivalent to starting the REPL and then executing `loadFile "name-of-file-to-load.egi"`.

```
$ egison -l name-of-file-to-load.egi
```

Executing a program in files

You can write any expressions at the top level of program. With `-t` option, the interpreter prints out the evaluation results of each expression written at the top level.

Note that the statements (such as definitions and `loadFile`) are not expressions, and hence not printed.

```
$ cat name-of-file-to-test.egi
x := 1
x + 2
```

(continues on next page)

(continued from previous page)

```
"This is the third line"

$ egison -t name-of-file-to-test.egi
3
"This is the third line"
```

Finally, simply passing a file name to `egison` executes the `main` function defined in the file. The `main` should be a unary (1 argument) function that returns an IO function. Command line arguments are given to the `main` function as a collection of string.

```
$ cat name-of-file-to-run.egi
main args :=
   print "Hello, world!"

$ egison name-of-file-to-run.egi
Hello, world!
```

1.2 Egison Quick Tour

This section introduces basic functionalities of Egison for pattern-match-oriented programming.

1.2.1 `matchAll` and matchers

Egison provides some syntactic constructs for expressing pattern matching. The most basic one among them is `matchAll`.

```
matchAll [1,2,3] as list something with
  | $x :: $xs -> (x, xs)
-- [(1,[2,3])]
```

A `matchAll` expression consists of the following elements.

- a **target** (`[1,2,3]` in the above example)
- a **matcher** (`list something`)
- more than one **match clauses** (`$x :: $xs -> (x, xs)`)

A match clause contains a **pattern** (`$x :: $xs` in the above example) and a **body** (`(x, xs)`). Just like the pattern matching in other programming languages, the `matchAll` expression attempts pattern matching of the target and the pattern, and if it succeeds, evaluates the body of the match clause.

The unique feature of the `matchAll` expression is twofold: (1) it returns a list, and (2) it takes additional argument called matchers.

(1) is for supporting pattern matching with multiple results. Since there can be multiple ways to match the pattern for the target data, the `matchAll` expression evaluates the body for all of these pattern-matching results and returns a list of the evaluation results. In the above example, the `::` is what we call a **cons pattern** which decomposes a list into the first element and the others. Because there is only one way to decompose the list `[1, 2, 3]` in this manner, the `matchAll` returns a singleton list.

The feature (2) realizes extensible pattern-matching algorithm and pattern polymorphism. Matcher is an Egison-specific object that retains pattern-matching algorithms. See *Ad-hoc polymorphism of patterns by matchers* for the description of pattern polymorphism.

Lines starting with -- are comments. In this tutorial, a line comment right after a program shows the execution result of the program.

We will explain more on the syntax of `matchAll`. A matcher is sandwitched between two keywords `as` and `with`. A `matchAll` expression can take multiple match clauses. Match clauses are precedented with `|`, which enhances the readability of program when a match clause occupy multiple lines. In a match clause, the pattern and the body is separated with `->`.

```
matchAll target as matcher with
| pattern1 -> body1
| pattern2 -> body2
...
```

When there is only one match clause, we can omit the `|` before the match clause.

```
matchAll target as matcher with pattern -> body
```

The following is an example of pattern matching with multiple results. `++` is called **join pattern**, which splits a list into two segments. The `matchAll` evaluates the body for every possible matching result of the join pattern.

```
matchAll [1,2,3] as list something with
| $hs ++ $ts -> (hs, ts)
-- [([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]
```

1.2.2 Non-linear pattern with value pattern and predicate pattern

`matchAll` gets even more powerful when combined with non-linear patterns. For example, the following non-linear pattern matches when the target collection contains a pair of identical elements.

```
matchAll [1,2,3,2,4,3] as list integer with
| _ ++ $x :: _ ++ #x :: _ -> x
-- [2,3]
```

Value patterns play an important role in representing non-linear patterns. A value pattern matches the target if the target is equal to the content of the value pattern. A value pattern is prepended with `#` and the expression after `#` is evaluated referring to the value bound to the pattern variables that appear on the left side of the patterns. As a result, for example, `$x :: #x :: _` is valid while `#x :: $x :: _` is invalid.

Let us show pattern matching for twin primes as a sample of non-linear patterns. A twin prime is a pair of prime numbers of the form $(p, p+2)$. `primes` is an infinite list of prime numbers which is defined in one of Egison standard libraries. This `matchAll` extracts all twin primes from this infinite list of prime numbers in order.

```
twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: #(p + 2) :: _ -> (p, p + 2)

take 8 twinPrimes
-- [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
```

There are cases that we might want to use more general predicates in patterns than equality. **Predicate patterns** are provided for such a purpose. A predicate pattern matches the target if the predicate returns true for the target. A predicate pattern is prepended with `?`, and a unary predicate follows after `?`.

```
twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: ?(\q -> q = p + 2) :: _ -> (p, p + 2)
```

1.2.3 Efficient pattern matching with backtracking

The pattern-matching algorithm inside Egison includes a backtracking mechanism for efficient non-linear pattern matching.

```
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ -> x
-- returns [] in O(n^2) time
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ ++ #x :: _ -> x
-- returns [] in O(n^2) time
```

The above expressions match a collection that consists of integers from 1 to n as a list of integers for enumerating identical pairs and triples, respectively. Since this target collection contains neither identical pairs nor triples, both expressions return an empty collection.

When evaluating the second expression, Egison interpreter does not try pattern matching for the second `#x` because pattern matching for the first `#x` always fails. Therefore, the time complexities of the above expressions are identical. The pattern-matching algorithm inside Egison is discussed in this paper in detail. .. TODO: write a manual page for algorithm

1.2.4 Ad-hoc polymorphism of patterns by matchers

Another merit of matchers, in addition to the extensibility of pattern-matching algorithms, is the **ad-hoc polymorphism of patterns**. The ad-hoc polymorphism of patterns allows us to use the same pattern constructors such as `::` and `++` for different matchers like `list` and `multiset`. It is important for non-free data types because some data are pattern-matched as various non-free data types at the different parts of a program. For example, a list can be pattern-matched as a multiset or a set. Polymorphic patterns reduce the number of names for pattern constructors.

In the following sample, a **collection** `[1,2,3]` is pattern-matched using different matchers with the same cons pattern. The "collection" is actually what we have been calling "list" so far. In Egison, collection refers to the sequential data that can be pattern-matched as lists, multisets or sets.

When we use `multiset` matcher, the cons pattern decomposes a collection into one element and the others ignoring the order of the elements. When we use `set` matcher, the right hand side of the cons pattern is matched with the original collection. This behavior comes from the idea that a set can be seen as a collection which contains infinitely many copies of each element.

```
matchAll [1,2,3] as list something with $x :: $xs -> (x,xs)
-- [(1,[2,3])]

matchAll [1,2,3] as multiset something with $x :: $xs -> (x,xs)
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]

matchAll [1,2,3] as set something with $x :: $xs -> (x,xs)
-- [(1,[1,2,3]),(2,[1,2,3]),(3,[1,2,3])]
```

1.2.5 Controlling the order of pattern matching

The `matchAll` expression is designed to enumerate all countably infinite pattern-matching results. For this purpose, users sometimes need to care about the order of pattern-matching results.

Let us start by showing a typical example. The `matchAll` expression below enumerates all pairs of natural numbers. We extract the first 8 elements with the `take` function. `matchAll` uses breadth-first search to traverse all the nodes in the reduction tree of pattern matching. .. TODO: Refer to the chapter of pattern-matching mechanism As a result, the order of the pattern-matching results is as follows.

```
take 8 (matchAll [1..] as set something with
        | $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(2,3),(3,2)]
```

The above order is suitable for traversing an infinite reduction tree. However, sometimes this order is not preferable. .. (see section 3.1.2 and section 3.4.1). `matchAllDFS`, which traverses a reduction tree in depth-first order, is provided for this reason.

```
take 8 (matchAllDFS [1..] as set something with
| $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]
```

For instance, think about defining `concat` with pattern matching. If we use `matchAll`, the outcome will be the alternation of the elements in the sublists, which is not what we expect of `concat`.

```
concat' xss := matchAll xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x

concat' [[1,2,3],[4,5,6],[7,8,9]]
-- [1, 2, 4, 3, 5, 7, 6, 8, 9]
```

To fix this, we should use `matchAllDFS` instead.

```
concat xss := matchAllDFS xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x

concat [[1,2,3],[4,5,6],[7,8,9]]
-- [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.2.6 Logical patterns: and-, or-, and not-patterns

Logical patterns such as **and-patterns**, **or-patterns** and **not-patterns** play an important role in enriching the expressivity of patterns.

An and-pattern `p1 & p2` pattern-matches when *both* of the patterns `p1` and `p2` succeeds in pattern matching. Similarly, an or-pattern `p1 | p2` pattern-matches when *either* of the patterns `p1` and `p2` succeeds in pattern matching. A not-pattern `!p` pattern-matches when the pattern `p` fails to pattern-match.

We start by showing pattern matching for prime triples as an example of and-patterns and or-patterns. A prime triple is a triple of primes of the form $(p, p+2, p+6)$ or $(p, p+4, p+6)$. The or-pattern `#(p + 2) | #(p + 4)` is used to match $p+2$ or $p+4$. The and-pattern `(#(p + 2) | #(p + 4)) & $m` binds the value matched by `(#(p + 2) | #(p + 4))` to a new variable `m`. This usage of and-pattern is similar to the as-pattern in Haskell.

```
primeTriples := matchAll primes as list integer with
  | _ ++ $p :: ((#(p + 2) | #(p + 4)) & $m) :: #(p + 6) :: _
  -> (p, m, p + 6)

take 6 primeTriples
-- [(5,7,11),(7,11,13),(11,13,17),(13,17,19),(17,19,23),(37,41,43)]
```

As an example of not-patterns, the following `matchAll` enumerates sequential pairs of prime numbers that are not twin primes. The not-pattern `!#(p + 2)` matches values other than $p+2$.

```
take 10 (matchAll primes as list integer with
         | _ ++ $p :: (!#(p + 2) & $q) :: _ -> (p, q))
-- [(2,3),(7,11),(13,17),(19,23),(23,29),(31,37),(37,41),(43,47),(47,53),(53,59)]
```

1.2.7 Loop Patterns

A loop pattern is a pattern construct for representing a pattern that repeats itself multiple times. It is an extension of Kleene star operator of regular expressions for general non-free data types.

Let us start by considering pattern matching for enumerating all combinations of two elements from a target collection. It can be written using `matchAll` as follows.

```
comb2 xs := matchAll xs as list something with
  | _ ++ $x_1 :: _ ++ $x_2 :: _ -> [x_1, x_2]

comb2 [1,2,3,4] -- [[1,2],[1,3],[2,3],[1,4],[2,4],[3,4]]
```

Egison allows users to append indices to a pattern variable as `$x_1` and `$x_2` in the above sample. They are called **indexed variables** and represent x_1 and x_2 in mathematical expressions. The expression after _ must be evaluated to an integer and is called an **index**. We can append as many indices as we want like `x_i_j_k`. When a value is bound to an indexed pattern variable `$x_i`, the system initiates an abstract map consisting of key-value pairs if `x` is not bound to a map, and bind it to `x`. If x is already bound to a map, a new key-value pair is added to this map.

Now, we generalize `comb2`. The loop patterns can be used for this purpose.

```
comb n xs := matchAll xs as list something with
  | loop $i                -- index variable
         (1, n)            -- index range
         (_ ++ $x_i :: ...) -- repeat pattern
         _                 -- final pattern
  -> map (\i -> x_i) [1..n]

comb 2 [1,2,3,4] -- [[1,2],[1,3],[2,3],[1,4],[2,4],[3,4]]
comb 3 [1,2,3,4] -- [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

A loop pattern consists of the following four elements.

- An **index variable** is a variable to hold the current repeat count.
- An **index range** is a tuple of an initial number and final number which specifies the range of the index variable.
- A **repeat pattern** is a pattern repeated when the index variable is in the index range.
- A **final pattern** is a pattern expanded when the index variable gets out of the index range.

Inside of the repeat patterns, we can use the ellipsis pattern `...`. The repeat pattern or the final pattern is expanded at the location of the ellipsis pattern. The repeat pattern is expanded replacing the ellipsis pattern incrementing the value of the index variable. For example, when `n` is 3, the above loop pattern is unfolded into as follows.

```
(loop $i (1, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: (loop $i (2, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: (loop $i (3, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: (loop $i (4, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: _
```

The repeat count of the loop patterns in the above example is constant. However, we can also write a loop pattern whose repeat count varies depending on the target by specifying a pattern instead of an integer as the final number. When the final number is a pattern, the ellipsis pattern is replaced with both the repeat pattern and the final pattern, and the repeat count when the ellipsis pattern is replaced with the final pattern is pattern-matched with that pattern. The following loop pattern enumerates all initial prefixes of the target collection.

```
matchAll [1,2,3,4] as list something with
| loop $i (1, $n) ($x_i :: ...) _ -> map (\i -> x_i) [1..n]
-- [[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

1.2.8 Sequential Patterns

The pattern-matching system of Egison processes patterns from left to right. However, there are some cases where we want to change this order, for example, to refer to a pattern variable bound in the right side of a pattern. **Sequential patterns** are provided for such cases.

Sequential patterns allow users to control the order of the pattern-matching process. A sequential pattern is represented as a list of patterns. Pattern matching is executed for each pattern in order. In the following sample, the target list is pattern-matched from the third, first, and second element in order.

```
matchAll [2,3,1,4,5] as list integer with
  | [    @    ::    @    :: $x :: _,
     (#(x + 1),     @    ),
               #(x + 2)]
  -> "Matched"
-- ["Matched"]
```

`@` that appears in a sequential pattern is called **later pattern variable**. The target data bound to later pattern variables are pattern-matched in the next sequence. When multiple later pattern variables appear, they are pattern-matched as a tuple in the next sequence.

Sequential patterns allow us to apply not-patterns for different parts of a pattern at the same time. For example, the following pattern matches when `xs` and `ys` have only one element in common. The use of the sequential pattern in this example allows us to first check that the two collections have at least one element in common, and then make sure that there is no more common element in the remaining part of the collections. Such combination of sequential patterns and not patterns is often useful when writing a mathematical algorithm.

```
singleCommonElem :=
  match (xs, ys) as (multiset eq, multiset eq) with
    | [($x :: @, #x :: @),
      !($y :: _, #y :: _)] -> True
```

Some readers might wonder if sequential patterns can be implemented using nested `matchAll` expressions. There are at least two reasons why it is impossible. First, a nested `matchAll` expression breaks breadth-first search strategy: the inner `matchAll` for the second result of the outer `matchAll` is executed only after the inner `matchAll` for the first result of the outer `matchAll` is finished. Second, a later pattern variable retains the information of not only a target but also a matcher. There are cases that the matcher of `matchAll` is a parameter passed as an argument of a function, and a pattern is polymorphic. Therefore, it is impossible to determine the matchers of inner `matchAll` expressions syntactically.

1.2.9 Pattern functions

It is sometimes the case that the same combination of patterns appears at multiple locations of a program. In such case, we can use **pattern functions** to give names to the combinations of patterns and avoid repetition.

A pattern function is a function which takes patterns as its argument and returns a pattern. Its syntax is similar to that of lambda functions except that it uses => instead of ->.

The `twin` in the following program is a pattern function and modularizes the double nested cons pattern. The argument of pattern functions are called **variable patterns**, which are `pat1` and `pat2` in the following case. Variable patterns must be prefixed with ~ when referred to in the body of pattern functions. This is necessary for distinguishing variable patterns from nullary pattern constructors.

```
twin := \pat1 pat2 => (~pat1 & $x) :: #x :: ~pat2

match [1, 1, 2, 3] as list integer with
```

(continues on next page)

(continued from previous page)

```
  | twin $n $ns -> [n, ns]
-- [1, [2, 3]]
```

1.2.10 Matcher compositions

All the matchers presented so far can be defined by users, except for the only built-in matcher `something`. Matchers are usually defined by the `matcher` expressions, but users can define matchers by composing the existing matchers. This way, we can for example define matchers for tuples of multisets and multisets of multisets.

First, we can define a matcher for tuples by a tuple of matchers. A tuple pattern is used for pattern matching using such a matcher. For example, we can define the intersect function using a matcher for tuples of two multisets.

```
intersect xs ys := matchAll (xs,ys) as (multiset eq, multiset eq) with
  | ($x :: _, #x :: _) -> x
```

`eq` is a user-defined matcher for data types for which equality is defined. When it is used, equality is checked for a value pattern. By passing a tuple matcher to a function that takes and returns a matcher, we can define a matcher for various non-free data types. For example, we can define a matcher for a graph as a set of edges as follows, where the nodes are represented by integers.

```
graph := multiset (integer, integer)
```

A matcher for adjacency graphs can also be defined. An adjacency graph is defined as a multiset of tuples of an integer and a multiset of integers.

```
adjacencyGraph := multiset (integer, multiset integer)
```

Egison provides a handy syntactic sugar for defining a matcher for algebraic data types, while it can also be defined with `matcher` expressions. For example, a matcher for binary trees can be defined using `algebraicDataMatcher`.

```
binaryTree a := algebraicDataMatcher
  | bLeaf a
  | bNode a (binaryTree a) (binaryTree a)
```

Matchers for algebraic data types and matchers for non-free data types also can be composed. For example, we can define a matcher for trees whose nodes have an arbitrary number of children whose order is ignorable.

```
tree a := algebraicDataMatcher
  | leaf a
  | node a (multiset (tree a))
```

1.3 Basics of I/O

This chapter explains how to use IO operation in Egison.

1.3.1 Hello World!

Let's start this tutorial by greeting the world. The following is the "Hello world" program in Egison.

```
-- Save this code as a "hello.egi" file
main args :=
  write "Hello, World!\n"
```

We can execute the above program as follows.

```
$ egison hello.egi
Hello, World!
```

Egison I/O works via a function named `main`. In main, we can use I/O functions such as `write`. (The list of I/O primitive functions is given in *List of Primitive Functions*.)

1.3.2 Command Line Arguments

Command line arguments are given to the `main` as its argument as a collection of strings.

For instance, assume the following program.

```
-- Save this code as a "args.egi" file
main args :=
  write (show args)
```

If you execute the following commands, you will see that the arguments are given to `main` as `args`.

```
$ egison args.egi
[]
$ egison args.egi We can write scripts in Egison
["We", "can", "write", "scripts", "in", "Egison"]
```

1.3.3 `do` expressions

To combine several I/O operations to one, we can use `do` expressions. The feature of `do` expressions is *serial* execution; the I/O functions in the `do` expressions are executed from the top in order. If you know Haskell, you probably notice that it is the same with the `do` expressions in Haskell.

```
-- Save this code as a "repl.egi" file
repl := do
  write "input: "
  flush ()
  let input := readLine ()
  write input
  print ""
  repl

main args := repl
```

Then, execute it as follow. Note that `write "input: "`, `flush ()`, `readLine ()` and `write input` are executed in the order.

```
$ egison repl.egi
input: Hi
Hi
input: Hello
Hello
```

(continues on next page)

(continued from previous page)

```
input: Repl
Repl
input: ^C
$
```

Check out *do expression* for more detail.

1.3.4 `io` expressions

We can use `io` expression to execute IO functions anywhere. For example, the following is a definition of the `pureRand` function in `lib/core/io.egi`.

```
pureRand s e := io rand s e
```

Check out *io expression* for more detail.

CHAPTER 2

Reference

2.1 Installation

There are two ways to install Egison: installing with package manager or installing from Haskell Platform. The former method is available for only Linux and Mac users, while the latter is available for every user.

	Install with package manager	Install from Haskell Platform
Linux	O (*yum* or *dpkg*)	O
MacOS	O (*Homebrew*)	O
Windows	X	O

2.1.1 Install with Package Manager

yum

```
$ sudo yum install https://git.io/egison.x86_64.rpm https://git.io/egison-tutorial.
↪x86_64.rpm
```

dpkg

```
$ wget https://git.io/egison.x86_64.deb https://git.io/egison-tutorial.x86_64.deb
$ sudo dpkg -i ./egison*.deb
```

Homebrew

```
$ brew update
$ brew tap egison/egison
$ brew install egison egison-tutorial
```

2.1.2 Install from Haskell Platform

1. Install Haskell Platform

To install Egison, you need to install Haskell Platform. This is because Egison is implemented in Haskell and distributed as a Hackage package.

If you use `apt-get`, execute the following commands.

```
$ sudo apt-get update
$ sudo apt-get install haskell-platform libncurses5-dev
```

Otherwise, download an installer from here.

2. Install Egison via Hackage

After you installed Haskell Platform, perform the following commands in the terminal.

```
$ cabal update
$ cabal install egison egison-tutorial
...
Installing executable(s) in /home/xxx/.cabal/bin
Registering egison-X.X.X...
```

When the installation is finished, there will be a message that tells the location of the installed binary. Add the path to the `$PATH` vairable so that your shell can find the `egison` command. For example, if you are using bash, run the following commands.

```
$ echo "PATH=\$PATH:/home/xxx/.cabal/bin" >> ~/.bashrc
$ source ~/.bashrc
```

2.2 Built-in Data

2.2.1 Character

Characters are enclosed in single quotes.

```
'a'
' '
'\n'
```

2.2.2 String

Strings are enclosed in double quotes.

```
"Hello, world"
```

2.2.3 Boolean

`True` and `False` are booleans.

2.2.4 Scalar Values

In Egison, numeric scalar values (except for floats) are treated as polynomials and represented in the mathematical canonical form.

Integer

```
1
0
-100
```

Rational number

```
1 / 3
4 / 6 ---> 2 / 3
```

Symbols

Unbound variables are interpreted as symbols.

```
> x + 1
x + 1
> f 2  -- uninterpreted functions
f 2
```

Mathematical expressions with symbols are automatically normalized into the normal form.

```
> (x + 1) ^ 2
x^2 + 2 * x + 1
```

Special symbols

Egison implements normalization algorithm for some of the common mathematical symbols.

`i` (imaginary unit):

```
> i * i
-1
> (1 + i)^2
2 * i
```

`sqrt` and `rt` (`sqrt n` denotes $\sqrt{n}$ and `rt m n` denotes $\sqrt[m]{n}$):

```
> (sqrt 2) ^ 2
2
> (rt 3 2) ^ 3
2
```

`sin` and `cos`:

```
> (sin x)^2 + (cos x)^2
1
```

2.2.5 Float

```
1.0
1e-1 ---> 0.1
2e3  ---> 2000.0
```

2.2.6 Inductive data

A variable starting with an upper-case letter is interpreted as a constructor of inductive data. You don't need to define the inductive data before using it.

```
Leaf
Node 1 Leaf (Node 2 Leaf Leaf)
```

2.2.7 Tuple (Multiple values)

A tuple is denoted as a sequence of elements enclosed in parentheses and separated by `,`. Tuples of single element cannot be written.

```
() -- zero-element tuple
(1, 2)
(2, "foo", True)
```

2.2.8 Collection

A collection is a sequence of elements that are enclosed in brackets and separated by `,`.

```
[]
[1]
[1, 2]
[1, 2, 3]
```

2.2.9 Tensor

A tensor is a sequence of elements enclosed in double brackets `[| |]` and separated by `,`. The i-th element of a tensor `t` can be retrieved by `t_i`. Note that it is 1-indexed.

```
t := [| 1, 2, 3, 4, 5 |]

t_1 ---> 1

-- The index can be any expression that evaluates to an integer.
t_(2+3) ---> 5

t_6 ---> Error: Tensor index out of bounds
```

You can get the shape of a tensor with `tensorShape`.

```
tensorShape [| 1, 2, 3, 4, 5 |] ---> [5]
```

Multi-dimensional tensors can be defined by nesting tensors.

```
[| [| 1, 2, 3 |], [| 4, 5, 6 |], [| 7, 8, 9 |] |]_1   ---> [| 1, 2, 3 |]
[| [| 1, 2, 3 |], [| 4, 5, 6 |], [| 7, 8, 9 |] |]_2_3 ---> 6
```

Egison prepares special syntax for tensors. See *Syntax for Tensor Computation* for detail.

2.2.10 Hash Maps

A hash map is a sequence of key-value pairs enclosed in double braces `{| |}`. The value of a key `k` in a hash map `h` can be retrieved by `h_k`. If the key is not included in the keys of the hash map, the result will be `undefined`.

```
{| (1, 11) (2, 12) (3, 13) (4, 14) (5, 15) |}_1 ---> 11
{| (1, 11) (2, 12) (3, 13) (4, 14) (5, 15) |}_4 ---> 14
{| (1, 11) (2, 12) (3, 13) (4, 14) (5, 15) |}_8 ---> undefined
```

2.2.11 IO Function

IO functions are functions that will yield IO operation when executed.

Any IO functions can be executed with *io expressions*.

```
print "foo" ---> #<io-function>
```

2.2.12 Port

A port has information of a file and its access mode (input/output). You can create a port with `openInputFile` or `openOutputFile`.

2.2.13 Undefined

`undefined` is a useful built-in data you can put where you have not written yet.

2.3 Basic Syntax

2.3.1 Top-level Expressions

Definition

You can bind an expression to a variable by connecting them with `:=`. When defining functions, the argument variable can be placed in the left hand side of the `:=`.

```
x := 1

-- The following two definitions are identical.
f := \x -> x + 1
f x := x + 1
```

Expression

You can also write arbitrary expressions at the top level of programs. These expressions are evaluated only when the `-t` option (see *-t / –test*) is specified.

```
-- definitions
x := 1
y := 2

-- A top-level expression which will evaluate to 3.
x + y
```

`load` and `loadFile`

We can load Egison libraries with `load`. To load your own program, call `loadFile` with a full-path or a relative path to the file.

```
-- Load Egison library.
load "lib/core/number.egi"

-- Load your program.
loadFile "myfile.egi"
```

Infix Declaration

(From version 4.0.4)

You can define your own infixes (binary operators) for expressions and patterns. In Egison, infix declaration consists of the following 4 parts.

- Associativity ... `infix` (non associative), `infixl` (left associative) or `infixr` (right associative)
- Infix type ... `expression` (for functions) or `pattern` (for pattern constructors)
- Priority of the infix
- Representation of infix

```
-- Define a right-associative infix '&&' of priority 5.
infixr expression 5 &&

-- Definition of the semantics of '&&'.
(&&) a b := match (a, b) as (eq, eq) with
              | (#True, #True) -> True
              | _              -> False

-- Define a left-associative infix '<>' of priority 7.
infixl pattern 7 <>

exampleMatcher := matcher
  | $ <> $ as (integer, integer) with
    | $x :: $y :: [] -> [(x, y)]
    | _              -> []

match [1, 2] as dummyMatcher with $x <> $y -> x + y
---> 3
```

2.3.2 Basic Expressions

Anonymous function

An anonymous function consists of two parts: arguments and a body. The arguments are written between `\` and `->`, and the body is written at the right of `->`.

```
-- A function of one argument.
\x -> x + 1

-- A function of two arguments.
\x y -> x + y

-- A function of no arguments.
\() -> 1

-- Function application
(\x y -> x + y) 3 7  ---> 10
```

The arguments can be simply aligned (separated with whitespace) or packed in a tuple. Namely, the following two notations are identical.

```
(\x y -> x + y)) 3 7    ---> 10
(\(x, y) -> x + y) 3 7  ---> 10
```

Anonymous parameter function

Egison has a shorthand notation for the anonymous function. In this syntax, the function body is prefixed with a `n#`, where `n` indicates the arity of the function. There must not be any spaces between the arity number `n` and the `#`. Also, the arguments are specified by numbers, where `%i` refers to the i-th argument.

This syntax is inspired by the anonymous function syntax of Clojure.

```
-- The followings are identical.
2#(%1 + %2)
\x y -> x + y
```

Section

Egison has a special syntax for the partial application of infix operators, which is inspired by the section notation of Haskell.

- `(+)` is desugared into `\x y -> x + y`
- `(+ 1)` is desugared into `\x -> x + 1`
- `(1 +)` is desugared into `\x -> 1 + x`

`let ... in` expression

A `let ... in` expression (or simply a `let` expression) locally binds expressions to variables. Bindings defined in a `let` expression cannot be referred to outside of the `let` expression.

```
let x := 1 in x + 1 ---> 2
```

You can write multiple bindings in a single `let` expression. Note that the head of the binding must be aligned vertically in order to be parsed correctly.

```
let x := 1
    y := 2
 in x + y
---> 3
```

The above expression can be written in a single line as follows. The bindings must be wrapped with `{ }` and separated with `;`.

```
let { x := 1 ; y := 2 } in x + y
```

Bindings in the same `let` expression can depend on each other. The bindings do not necessarily be aligned in the order of dependency.

```
let y := x  -- 'x' is defined in the next binding
    x := 1
 in y
---> 1

-- We can even define mutual-recursive functions.
let isOdd n := if n = 0 then False else isEven (n - 1)
    isEven n := if n = 0 then True else isOdd (n - 1)
 in isOdd 5
---> True
```

As a result, note the following behavior.

```
x := 3

let y := x
    x := 1
 in y
---> 1 (not 3)
```

`where` expression

`where` is a syntax sugar for the above `let` expression. Unlike the `let` expression, the bindings in `where` expressions come after the body expression.

For example, the following two expressions are identical.

```
-- local bindings with `where`
expression
  where
    x1 := expr1
    x2 := expr2

-- local bindings with `let`
let x1 := expr1
    x2 := expr2
 in expression
```

`if` expression

It is the ordinary `if` expression. The guard expression (the one right after `if`) must be evaluated to a boolean (`True` or `False`).

```
if True then "Yes" else "No"  ---> "Yes"
if False then "Yes" else "No" ---> "No"
```

`do` expression

A `do` expression can group several IO functions into one IO function. You can bind expressions to values with `let` in the `do` expression as well. Every lines in the `do` block must either be an expression that evaluates to an IO function or a `let` binding. Note that all the lines in the `do` block must be aligned vertically.

```
repl := do
  write ">>> "
  flush ()
  let line := readLine ()
  write line
  flush ()
  repl
```

A `do` expression can be written in one line as follows. The expressions needs to be wrapped with `{ }` and separated by `;`.

```
do { print "foo" ; print "bar" ; print "baz" }
```

The last statement in a `do` block must be an expression. The last expression in a `do` block is interpreted as the evaluation result of the `do` expression.

```
> io do { return 1; return 2; return 3 }
3
```

`io` expression

An `io` expression takes an IO function and executes it. This is similar to the `unsafePerformIO` in Haskell.

```
> io print "hoge"
hoge
()
```

`seq` expression

This expression is inspired by the `seq` function in Haskell.

A `seq` expression takes two arguments. The first argument of `seq` is strictly evaluated. The most popular use case of seq is in the definition of the foldl function.

```
foldl $fn $init $ls :=
  match ls as list something with
    | [] -> init
    | $x :: $xs ->
      let z := fn init x
       in seq z (foldl fn z xs)
```

2.4 Pattern Matching

2.4.1 Expressions for pattern matching

`matchAll` expression

A `matchAll` expression takes a target, a matcher and one or more match clauses. It tries pattern matching for all match clauses, and returns a collection of the evaluation result of the body for all successful result of pattern matching.

```
matchAll [1, 2, 3] as list integer with
| $x :: $xs -> (x, xs)
---> [(1, [2, 3])]

matchAll [1, 2, 3] as multiset integer with
| $x :: $xs -> (x, xs)
---> [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]

matchAll [1, 2, 3] as set integer with
| $x :: $xs -> (x, xs)
---> [(1, [1, 2, 3]), (2, [1, 2, 3]), (3, [1, 2, 3])]
```

When none of the match clauses successfully pattern-matches, `matchAll` returns an empty collection `[]`.

```
matchAll [] as list integer with
| $x :: $xs -> (x, xs)
---> []
```

You can write more than one match clauses. In that case, every match clause must start with `|` and the `|` of all match clauses must be vertically aligned.

```
matchAll [1, 2, 3] as multiset integer with
| [] -> -1
| $x :: $xs -> x
```

When there is only one match clause, the `|` can be omitted.

```
matchAll [1, 2, 3] as multiset integer with $x :: _ -> x
```

`match` expression

`match` expressions are similar to `matchAll` expressions except that it returns only one value. In fact, the return value of a `match` expression is defined as the first element of the return value of its corresponding `matchAll` expression.

```
match [1, 2, 3] as multiset integer with
| $x :: $xs -> (x, xs)
---> (1, [2, 3])
```

When none of the match clauses successfully pattern-matches, it will raise an error.

```
match [1, 2, 3] as multiset integer with
| [] -> "OK"
---> Failed pattern match in: <stdin>
```

`\matchAll` and `\match`

`\matchAll` and `\match` are handy syntax sugar for the combination of anonymous function and `matchAll`/`match` expressions.

The syntax of `\matchAll` expression is similar to that of `matchAll` except that it doesn't need the target. A `\matchAll` expression is desugared into an anonymous function whose body is `matchAll` and whose argument is the target of `matchAll`.

For example,

```
\matchAll as matcher with
  | pattern1 -> expr1
  | pattern2 -> expr2
```

is desugared into the following expression.

```
\x ->
  matchAll x as matcher with
    | pattern1 -> expr1
    | pattern2 -> expr2
```

The semantics of `\match` is similar.

`matchAllDFS` and `matchDFS`

`matchAllDFS` and `matchDFS` are variants of `matchAll` and `match`, respectively. See *Controlling the order of pattern matching* for the description.

Pattern functions

A pattern function is a function that takes patterns and returns a pattern. Pattern functions allows us to reuse useful combination of patterns.

The syntax of pattern function is similar to that of *anonymous function* except that it uses double arrow => instead of the sigle arrow ->. Also, the argument pattern must be prefixed with a ~ in the body of the pattern function. This is to distinguish the argument with nullary pattern constructor.

The application of pattern functions is written in the same manner as the application of pattern constructors.

```
-- Defining a pattern function 'twin'
twin := \ pat1 pat2 => ($pat & ~pat1) :: #pat :: ~pat2

matchAll [1, 2, 1, 3] as multiset integer with twin $n _ -> n
---> [1, 1]

matchAll [2, 2, 1, 3] as multiset integer with _ :: twin #1 _ -> True
---> []
```

Like anonymous functions, a pattern function has lexical scope for the pattern variables. Therefore, bindings for pattern variables in the argument patterns and the body of pattern functions don't conflict.

2.4.2 Patterns

Wildcard pattern

Wildcard patterns are denoted by _. It can match with any values and the matched value will be discarded.

```
match [1, 2, 3] as list something with
| _ -> "OK"
---> "OK"
```

Pattern variable

We can bind values to variables in pattern matching with pattern variables. It is denoted as a variable prefixed with `$`. Any object matches pattern variables and the variable is locally bound to the object.

```
match True as bool with
| $x -> x
---> True

match [1, 2, 3] as list integer with
| $x :: $xs -> (x, xs)
---> (1, [2, 3])
```

Indexed pattern variable

Indexed pattern variables `$x_n` (`n` denotes integers) are special pattern variables. When an indexed pattern variable `$x_n` appears in the pattern, Egison creates a *hash map* and binds it to the variable `x`. An object matched to `$x_i` is associated with the key `i` in the hash `x`.

```
match 1 as something with $x_1 -> x
---> {| (1, 1) |}

match [1, 2, 3] as list integer with $x_1 :: $x_2 -> x
---> {| (1, 1), (2, [2, 3]) |}
```

Inductive pattern

Inductive pattern is an analogy of *inductive data*. An inductive pattern consists of a **pattern constructor** and multiple (zero or more) argument patterns. The names and behaviors of pattern constructors are defined by matchers.

In the following example, `snoc` is a pattern constructor defined in the `list` matcher, and `$x` and `$xs` is applied to the pattern constructor.

```
matchAll [1, 2, 3] as list integer with snoc $x $xs -> (x, xs)
---> (3, [1, 2])
```

The nil pattern `[]` and the pattern infixes such as `::` and ++ are also implemented as pattern constructors.

Value pattern

A value pattern is written as `#expr`, where `expr` can be any expression. An object `obj` can match a value pattern `#expr` only if the evaluation result of `obj` is equal to that of `expr`. This equality is defined by the given matcher.

```
match 1 as integer with
| #1 -> OK
| _  -> KO
---> OK

match 0 as integer with
| #1 -> OK
| _  -> KO
---> KO

match [1, 2, 3] as list integer with
| #[1, 2, 3] -> OK
---> OK

match [1, 2, 3] as multiset integer with
| #[2, 1, 3] -> OK
---> OK
```

Predicate pattern

A predicate pattern is a pattern that matches with an object when it satisfies the predicate following `?`. The expression following `?` should be a unary function that returns a boolean.

```
matchAll [1..6] as list integer with
| $xs ++ ?(< 4) :: $ys -> xs ++ ys
---> [[2, 3, 4, 5, 6], [1, 3, 4, 5, 6], [1, 2, 4, 5, 6]]

matchAll [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377] as multiset integer␣
↪with
| ?(\x -> modulo x 2 == 0) & $x -> x
---> [2, 8, 34, 144]
```

And-pattern

An and-pattern `p1 & p2` is a pattern that matches the object if and only if both of the pattern `p1` and `p2` are matched.

```
match [1, 3, 2] as list integer with
| (#1 :: _) & snoc #2 _ -> OK
| _                    -> KO
---> OK
```

We can use and-patterns like as-patterns in Haskell. For example, a pattern `(_ :: _) & $xs` matches with any non-empty collections and binds it to the variable `xs`.

```
match [1, 2] as list integer with
| (_ :: _) & $xs -> xs
---> [1, 2]

match [] as list integer with
| (_ :: _) & $xs -> xs
---> pattern match failure
```

Or-pattern

An or-pattern `p1 | p2` matches with the object if the object matches with `p1` or `p2`.

```
match [1, 3, 3] as list integer with
| (#1 :: _) | snoc #2 _ -> OK
| _                     -> KO
---> OK
```

Not-pattern

A not-pattern `!p` matches with the object if the object does not match the pattern `p`.

```
match 1 as integer with !#2 -> True
---> True

-- Returns True if and only if the collection does not contain 1
f :=
  \match as multiset integer with
    | !(#1 :: _) -> True
    | _          -> False

-- Returns True if and only if the collection has an element other than 1
g :=
  \match as multiset integer with
    | !#1 :: _ -> True
    | _        -> False

f [2, 3, 4] ---> True
f [1, 2, 3] ---> False
g [1, 2, 3] ---> True
g [1, 1, 1] ---> False
```

Sequential pattern

See *Sequential Patterns* in the tutorial.

Loop pattern

See *Loop Patterns* in the tutorial.

Let pattern

A let pattern allows binding expressions to variables inside the pattern. The variables bound in the `let` pattern can be used in the body of the `let` pattern.

```
f x :=
  match x as multiset integer with
  | let n := length x in #n :: #n :: _ -> True
  | _                                  -> False

f [1, 2, 2] ---> False
```

(continues on next page)

(continued from previous page)

```
f [3, 3, 2] ---> True
f [1, 2, 3, 4] ---> False
f [1, 4, 3, 4] ---> True
```

2.4.3 Matchers

`something` matcher

`something` is the only built-in matcher. Only variable pattern and wildcard patterns can be used for `something` matcher; it does not decompose the target object.

```
match [1, 2, 3] as something with $x -> x ---> [1, 2, 3]
match [1, 2, 3] as something with _  -> True ---> True
match [1, 2, 3] as something with $x :: _  -> x ---> Error
```

Defining matcher with `matcher` expression

This subsection describes how to define a matcher with `matcher` expression.

Let's think about defining a matcher `unorderedIntegerPair`, which matches with a tuple of 2 integers ignoring the order.

```
matchAll (1, 2) as unorderedIntegerPair with pair $a $b -> (a, b)
---> [(1, 2), (2, 1)]
```

This `unorderedIntegerPair` matcher can be defined as follows.

```
unorderedIntegerPair :=
  matcher
    | pair $ $ as (integer, integer) with
      | ($x, $y) -> [(x, y), (y, x)]
    | $ as something with
      | $tgt -> [tgt]
```

Line 3 and 4 corresponds with the case where we want to decompose the tuple, and line 5 and 6 is for the case where we don't want to. The expression `pair $ $` in line 3 is a **primitive pattern pattern** (pattern for patterns) and it defines a pattern constructor named `pair`, which enables the pattern expression like `pair $a $b`. The following `(integer, integer)` indicates that the both of matched 2 terms should be recursively pattern-matched by using `integer` matcher. The expression `($x, $y) -> [(x, y), (y, x)]` in line 4 defines the correspondense between the syntactic representation of the target data and pattern matching results. The `($x, $y)` in line 4 is called **primitive data pattern**. In the example above, the target data `(1, 2)` is *syntactically* matched with `($x, $y)`, making the variable `x` bound to `1` and `y` to `2`. As a result, the pattern matching result (specified with `[(x, y), (y, x)]`) will be `[(1, 2), (2, 1)]`. Then, variable `a` and `b` in the pattern expression `pair $a $b` are bound to one of the pattern matching result. Since it is a `matchAll` expression, this binding enumrates for the entire results, meaning that the first `a` is bound to `1` and `b` to `2`, and secondly `a` to `2` and `b` to `1`.

This `unorderedIntegerPair` matcher only works for integer tuples; however, we can make it "polymorphic" by making it a function that takes matchers and returns a matcher. For example, `unorderedPair` for an arbitrary matcher can be defined as follows:

```
unorderedPair m :=
  matcher
```

(continues on next page)

(continued from previous page)

```
    | pair $ $ as (m, m) with
      | ($x, $y) -> [(x, y), (y, x)]
    | $ as something with
      | $tgt -> [tgt]

-- Examples
match ([1, 2], [3, 4]) as unorderedPair (multiset integer) with
| pair (#4 :: _) _ -> True
---> True
```

`algebraicDataMatcher` expression

`algebraicDataMatcher` is a convenient syntax sugar for defining normal matchers, which decompose data accordingly to their data structure. For example, the following code defines a matcher for terms in untyped lambda calculus. The first identifiers in each line of the `algebraicDataMatcher` (`var`, `abs` and `app`) must start with a lower case alphabet.

```
term :=
  algebraicDataMatcher
    | var string       -- variable
    | abs string term  -- lambda abstraction
    | app term term    -- application
```

The above definition is desugared into the following one:

```
term :=
  matcher
    | var $ as string with
      | Var $x -> [x]
      | _      -> []
    | abs $ $ as (string, term) with
      | Abs $x $t -> [(x, t)]
      | _         -> []
    | app $ $ as (term, term) with
      | App $s $t -> [(s, t)]
      | _         -> []
    | $ as something with
      | $tgt -> [tgt]
```

2.5 Symbolic Computation

2.5.1 Syntax for Symbolic computation

`withSymbols` expression

`function` expression

Quoted expression

2.6 Tensor Computation

2.6.1 Syntax for Tensor Computation

`generateTensor` expression

`contract` expression

`tensorMap` expression

`tensorMap2` expression

`transpose` expression

`subrefs`, `suprefs` and `userRefs`

Wedge application expression

2.7 List of Libraries

2.7.1 List of core libraries

lib/core/assoc.egi

toAssoc

```
toAssoc [x, x, y, z] ---> [(x, 2), (y, 1), (z, 1)]
toAssoc [x, y, x] ---> [(x, 1), (y, 1), (x, 1)]
```

fromAssoc

```
fromAssoc [(x, 2), (y, 1)] ---> [x, x, y]
```

assocList

```
matchAll [(x, 2), (y, 1)] as assocList something with
  | $a :: _ -> a
---> [x]
matchAll [(x, 3), (y, 2), (z, 1)] as assocList something with
  | ncons $a #2 $r -> (a, r)
---> [(x, [(x, 1), (y, 2), (z, 1)])]
matchAll [(x, 1), (y, 2), (z, 3)] as assocList something with
  | ncons $a #2 $r -> (a, r)
---> []
matchAll [(x, 3), (y, 2), (z, 1)] as assocList something with
  | ncons $a $n $r -> (a, n, r)
---> [(x, 3, [(y, 2), (z, 1)])]
matchAll [(x, 3), (y, 2), (z, 1)] as assocList something with
  | ncons $a $n $r -> (a, n, r)
---> [(x, 3, [(y, 2), (z, 1)])]
```

assocMultiset

```
matchAll [(x, 2), (y, 1)] as assocMultiset something with
  | $a :: _ -> a
---> [x, y]
matchAll [(x, 3), (y, 2), (z, 1)] as assocMultiset something with
  | ncons #z $n $r -> (n, r)
---> [(1, [(x, 3), (y, 2)])]
matchAll [(x, 3), (y, 2), (z, 1)] as assocMultiset something with
  | ncons $a #2 $r -> (a, r)
---> [(x, [(x, 1), (y, 2), (z, 1)]), (y, [(x, 3), (z, 1)])]
matchAll [(x, 3), (y, 2), (z, 1)] as assocMultiset something with
  | ncons #y #1 $r -> r
---> [[(x, 3), (y, 1), (z, 1)]]
matchAll [(x, 3), (y, 2), (z, 1)] as assocMultiset something with
  | ncons $a $n $r -> (a, n, r)
---> [(x, 3, [(y, 2), (z, 1)]), (y, 2, [(x, 3), (z, 1)]), (z, 1, [(x, 3), (y,
↪2)])]
```

AC.intersect

```
AC.intersect [(x, 2), (y, 1)] [(x, 1), (y, 2)]
---> [(x, 1), (y, 1)]
AC.intersect [(x, 2), (y, 2)] [(x, 1), (y, 1)]
---> [(x, 1), (y, 1)]
AC.intersect [(x, 1), (y, 1)] [(x, 2), (y, 2)]
---> [(x, 1), (y, 1)]
```

lib/core/base.egi

id

```
id 1 ---> 1
```

fst

```
fst (1, 2) ---> 1
```

snd

```
snd (1, 2) ---> 2
```

compose

```
(compose fst snd) ((1, 2), 3) ---> 2
```

eqAs

```
eqAs integer 1 1 ---> True
```

and

```
[True && True, True && False, False && True, False && False]
---> [True, False, False, False]
```

or

```
[True || True, True || False, False || True, False || False]
---> [True, True, True, False]
```

not

```
[not True, not False] ---> [False, True]
```

lib/core/collection.egi

nth

```
nth 1 [1, 2, 3] ---> 1
```

take

```
take 2 [1, 2, 3] ---> [1, 2]
```

drop

```
drop 2 [1, 2, 3] ---> [3]
```

takeAndDrop

```
takeAndDrop 2 [1, 2, 3] ---> ([1, 2], [3])
```

takeWhile

```
takeWhile (< 10) primes ---> [2, 3, 5, 7]
```

head

```
head [1, 2, 3] ---> 1
```

tail

```
tail [1, 2, 3] ---> [2, 3]
```

last

```
last [1, 2, 3] ---> 3
```

init

```
init [1, 2, 3] ---> [1, 2]
```

uncons

```
uncons [1, 2, 3] ---> (1, [2, 3])
```

unsnoc

```
unsnoc [1, 2, 3] ---> ([1, 2], 3)
```

isEmpty

```
isEmpty [] ---> True
isEmpty [1] ---> False
```

length

```
length [1, 2, 3] ---> 3
```

map

```
map (* 2) [1, 2, 3] ---> [2, 4, 6]
```

map2

```
map2 (*) [1, 2, 3] [10, 20, 30] ---> [10, 40, 90]
```

filter

```
filter (\n -> n % 2 = 1) [1, 2, 3] ---> [1, 3]
```

zip

```
zip [1, 2, 3] [10, 20, 30]
---> [(1, 10), (2, 20), (3, 30)]
```

lookup

```
lookup 2 [(1, 10), (2, 20), (3, 30)] ---> 20
```

foldr

```
foldr (\n ns -> n :: ns) [] [1, 2, 3] ---> [1, 2, 3]
```

foldl

```
foldl (\ns n -> n :: ns) [] [1, 2, 3] ---> [3, 2, 1]
```

scanl

```
scanl (*) 2 [2, 2, 2] ---> [2, 4, 8, 16]
```

concat

```
concat [[1, 2], [3, 4, 5]] ---> [1, 2, 3, 4, 5]
```

reverse

```
reverse [1, 2, 3] ---> [3, 2, 1]
```

intersperse

```
intersperse [0] [[1, 2], [3, 3], [4], []]
---> [[1, 2], [0], [3, 3], [0], [4], [0], []]
```

intercalate

```
intercalate [0] [[1, 2], [3, 3], [4], []]
---> [1, 2, 0, 3, 3, 0, 4, 0]
```

split

```
split [0] [1, 2, 0, 3, 3, 0, 4, 0]
---> [[1, 2], [3, 3], [4], []]
```

splitAs

```
splitAs integer [0] [1, 2, 0, 3, 3, 0, 4, 0]
---> [[1, 2], [3, 3], [4], []]
```

findCycle

```
findCycle [1, 3, 4, 5, 2, 7, 5, 2, 7, 5, 2, 7]
---> ([1, 3, 4], [5, 2, 7])
```

repeat

```
take 5 (repeat [1, 2, 3]) ---> [1, 2, 3, 1, 2]
```

repeat1

```
take 5 (repeat1 2) ---> [2, 2, 2, 2, 2]
```

all

```
all (= 1) [1, 1, 1] ---> True
all (= 1) [1, 1, 2] ---> False
```

any

```
any (= 1) [0, 1, 0] ---> True
any (= 1) [0, 0, 0] ---> False
```

from

```
take 3 (from 2) ---> [2, 3, 4]
```

between

```
between 2 5 ---> [2, 3, 4, 5]
```

add

```
add 1 [2, 3] ---> [2, 3, 1]
add 1 [1, 2, 3] ---> [1, 2, 3]
```

addAs

```
addAs integer 1 [2, 3] ---> [2, 3, 1]
addAs integer 1 [1, 2, 3] ---> [1, 2, 3]
```

deleteFirst

```
deleteFirst 2 [1, 2, 3, 2] ---> [1, 3, 2]
```

deleteFirstAs

```
deleteFirstAs integer 2 [1, 2, 3, 2] ---> [1, 3, 2]
```

delete

```
delete 2 [1, 2, 3, 1, 2, 3] ---> [1, 3, 1, 3]
```

deleteAs

```
deleteAs integer 2 [1, 2, 3, 1, 2, 3]
---> [1, 3, 1, 3]
```

difference

```
difference [1, 2, 3] [1, 3] ---> [2]
```

differenceAs

```
differenceAs integer [1, 2, 3] [1, 3] ---> [2]
```

union

```
union [1, 2, 3] [1, 3, 4] ---> [1, 2, 3, 4]
```

unionAs

```
unionAs integer [1, 2, 3] [1, 3, 4] ---> [1, 2, 3, 4]
```

intersect

```
intersect [1, 2, 3] [1, 3, 4] ---> [1, 3]
```

intersectAs

```
intersectAs integer [1, 2, 3] [1, 3, 4] ---> [1, 3]
```

member

```
member 1 [1, 3, 1, 4] ---> True
member 2 [1, 3, 1, 4] ---> False
```

memberAs

```
memberAs integer 1 [1, 3, 1, 4] ---> True
memberAs integer 2 [1, 3, 1, 4] ---> False
```

count

```
count 1 [1, 3, 1, 4] ---> 2
```

countAs

```
countAs integer 1 [1, 3, 1, 4] ---> 2
```

frequency

```
frequency [1, 3, 1, 4] ---> [(1, 2), (3, 1), (4, 1)]
```

frequencyAs

```
frequencyAs integer [1, 3, 1, 4]
---> [(1, 2), (3, 1), (4, 1)]
```

unique

```
unique [1, 2, 3, 2, 1, 4] ---> [1, 2, 3, 4]
```

uniqueAs

```
uniqueAs integer [1, 2, 3, 2, 1, 4] ---> [1, 2, 3, 4]
```

lib/core/io.egi

print Prints the given string and a newline to the standard output.

```
> io print "foo"
foo
()
```

printToPort A variant of `print`. The output is written to the given port.

```
printToPort outport "foo"
```

display Prints the given string to the standard output.

```
> io display "foo"
foo()
```

displayToPort A variant of `display`. The output is written to the given port.

```
displayToPort outport "foo"
```

eachLine Repeat reading a single line from the standard input and applying the given function to it.

```
-- Repeats reading one line and printing it out
eachLine print
```

eachLineFromPort A variant of `eachLine`. The input is read from the given port.

```
eachLineFromPort inport print
```

eachFile Takes a collection of file names (in string) and a function, and apply the function for each line in each of the files.

```
eachFile ["in1.txt", "in2.txt"] print
```

each This function corresponds to the `mapM_` in Haskell.

```
each print ["foo", "bar"]
```

debug Prints out the argument value and returns it. This function is useful for debugging.

```
debug x
```

debug2 A variant of `debug`. The first argument is a string.

```
debug2 "x = " x
```

lib/core/maybe.egi

maybe

```
matchAll Just 1 as maybe integer with
  | just $x -> x
  | nothing -> "error"
---> [1]
matchAll Nothing as maybe integer with
  | just _ -> "error"
  | nothing -> True
---> [True]
```

lib/core/number.egi

nats

```
take 10 nats ---> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

nats0

```
take 10 nats0 ---> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

odds

```
take 10 odds ---> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

evens

```
take 10 evens
---> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

primes

```
take 10 primes
---> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

divisor

```
divisor 10 5 ---> True
```

findFactor

```
findFactor 1 ---> 1
findFactor 35 ---> 5
findFactor 100 ---> 2
```

pF

```
pF 1 ---> []
pF 3 ---> [3]
pF 100 ---> [2, 2, 5, 5]
```

isOdd

```
isOdd 3 ---> True
isOdd 4 ---> False
```

isEven

```
isEven 4 ---> True
isEven 5 ---> False
```

isPrime

```
isPrime 17 ---> True
isPrime 18 ---> False
```

perm

```
perm 5 2 ---> 20
```

comb

```
comb 5 2 ---> 10
```

nAdic

```
nAdic 10 123 ---> [1, 2, 3]
nAdic 2 10 ---> [1, 0, 1, 0]
```

rtod

```
2#(%1, take 10 %2) (rtod (6 / 35))
---> (0, [1, 7, 1, 4, 2, 8, 5, 7, 1, 4])
```

rtod'

```
rtod' (6 / 35) ---> (0, [1], [7, 1, 4, 2, 8, 5])
```

showDecimal

```
showDecimal 10 (6 / 35) ---> "0.1714285714"
```

showDecimal'

```
showDecimal' (6 / 35) ---> "0.1 714285 ..."
```

lib/core/order.egi

compare

```
compare 10 10 ---> Equal
compare 11 10 ---> Greater
compare 10 11 ---> Less
```

min

```
min 20 5 ---> 5
```

minimum

```
minimum [20, 5, 12] ---> 5
```

min/fn

```
min/fn compare [10, 20, 5, 20, 30] ---> 5
```

max

```
max 5 30 ---> 30
```

maximum

```
maximum [5, 30, 23] ---> 30
```

max/fn

```
max/fn compare [10, 20, 5, 20, 30] ---> 30
```

sort

```
sort [10, 20, 5, 20, 30] ---> [5, 10, 20, 20, 30]
```

sort/fn

```
sort/fn compare [10, 20, 5, 20, 30]
---> [5, 10, 20, 20, 30]
```

lib/core/random.egi

R.multiset

```
matchAll [1, 2] as R.multiset integer with
| $n :: $ns -> (n, ns)
---> [(1, [2]), (2, [1])] or [(2, [1]), (1, [2])]

matchAll [1, 2] as R.multiset integer with
| #1 :: $ns -> ns
---> [[2]]
```

R.set

```
matchAll [1, 2] as R.set integer with
| $n :: $ns -> (n, ns)
---> [(1, [1, 2]), (2, [1, 2])] or [(2, [1, 2]), (1, [1, 2])]

matchAll [1, 2] as R.set integer with
| #1 :: $ns -> ns
---> [[1, 2]]
```

pureRand

```
pureRand 1 6 ---> 1, 2, 3, 4, 5 or 6
```

randomize

```
randomize [1, 2, 3]
---> [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2] or [3, 2, 1]
```

R.between

```
R.between 1 3
---> [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2] or [3, 2, 1]
```

R.uncons

```
R.uncons [1, 2]
---> (1, [2]) or (2, [1])
```

R.head

```
R.head [1, 2]
---> 1 or 2
```

R.tail

```
R.tail [1, 2]
---> [2] or [1]
```

lib/core/string.egi

S.isEmpty

```
S.isEmpty "" ---> True
S.isEmpty "Egison" ---> False
```

S.head

```
S.head "Egison" ---> 'E'
```

S.tail

```
S.tail "Egison" ---> "gison"
```

S.last

```
S.last "Egison" ---> 'n'
```

S.map

```
S.map id "Egison" ---> "Egison"
```

S.length

```
S.length "Egison" ---> 6
```

S.split

```
S.split "," "Lisp,Haskell,Egison"
---> ["Lisp", "Haskell", "Egison"]
```

S.append

```
S.append "Egi" "son" ---> "Egison"
```

S.concat

```
S.concat ["Egi", "son"] ---> "Egison"
```

S.intercalate

```
S.intercalate "," ["Lisp", "Haskell", "Egison"]
---> "Lisp,Haskell,Egison"
```

C.between

```
C.between 'a' 'c' ---> ['a', 'b', 'c']
```

C.isBetween

```
C.isBetween 'a' 'c' 'b' ---> True
```

isAlphabet

```
isAlphabet 'a' ---> True
```

isAlphabetString

```
isAlphabetString "Egison" ---> True
```

upper-case

```
upperCase 'e' ---> 'E'
```

lower-case

```
lowerCase 'E' ---> 'e'
```

2.7.2 List of mathematical libraries

2.8 List of Primitive Functions

The following is the list of primitive (built-in) functions.

2.8.1 Pure primitive functions

numerator

```
numerator (13 / 21) ---> 13
```

denominator

```
denominator (13 / 21) ---> 21
```

modulo

```
modulo (-21) 13 ---> 5
```

quotient

```
quotient (-21) 13 ---> -1
```

remainder

```
remainder (-21) 13 ---> -8
```

neg

```
neg (-89) ---> 89
```

abs

```
abs 0 ---> 0
abs 15 ---> 15
abs (-89) ---> 89
```

lt

```
0.1 < 1.0 ---> True
1.0 < 0.1 ---> False
1.0 < 1.0 ---> False
```

lte

```
0.1 <= 1.0 ---> True
1.0 <= 0.1 ---> False
1.0 <= 1.0 ---> True
```

gt

```
0.1 > 1.0 ---> False
1.0 > 0.1 ---> True
1.0 > 1.0 ---> False
```

gte

```
0.1 >= 1.0 ---> False
1.0 >= 0.1 ---> True
1.0 >= 1.0 ---> True
```

round

```
round 3.1 ---> 3
round 3.7 ---> 4
round (-2.2) ---> -2
round (-2.7) ---> -3
```

floor

```
floor 3.1 ---> 3
floor 3.7 ---> 3
floor (-2.2) ---> -3
floor (-2.7) ---> -3
```

ceiling

```
ceiling 3.1 ---> 4
ceiling 3.7 ---> 4
ceiling (-2.2) ---> -2
ceiling (-2.7) ---> -2
```

truncate

```
truncate 3.1 ---> 3
truncate 3.7 ---> 3
truncate (-2.2) ---> -2
truncate (-2.7) ---> -2
```

sqrt

```
sqrt 4 ---> 2
sqrt 4.0 ---> 2.0
sqrt (-1) ---> i
```

exp

```
exp 1 ---> e
exp 1.0 ---> 2.718281828459045
exp (-1.0) ---> 0.36787944117144233
```

log

```
log e ---> 1
log 10.0 ---> 2.302585092994046
```

sin

```
sin 0.0 ---> 0.0
```

cos

```
cos 0.0 ---> 1.0
```

tan

```
tan 0.0 ---> 0.0
```

asin

```
asin 0.0 ---> 0.0
```

acos

```
acos 1.0 ---> 0.0
```

atan

```
atan 0.0 ---> 0.0
```

sinh

```
sinh 0.0 ---> 0.0
```

cosh

```
cosh 0.0 ---> 1.0
```

tanh

```
tanh 0.0 ---> 0.0
```

asinh

```
asinh 0.0 ---> 0.0
```

acosh

```
acosh 1.0 ---> 0.0
```

atanh

```
atanh 0.0 ---> 0.0
```

itof

```
itof 4 ---> 4.0
itof (-1) ---> -1.0
```

rtof

```
rtof (3 / 2) ---> 1.5
rtof 1 ---> 1.0
```

ctoi

```
ctoi '1' ---> 49
```

itoc

```
itoc 49 ---> '1'
```

pack

```
pack [] ---> ""
pack ['E', 'g', 'i', 's', 'o', 'n'] ---> "Egison"
```

unpack

```
unpack "Egison" ---> ['E', 'g', 'i', 's', 'o', 'n']
unpack "" ---> []
```

unconsString

```
unconsString "Egison" ---> ('E', "gison")
```

lengthString

```
lengthString "" ---> 0
lengthString "Egison" ---> 6
```

appendString

```
appendString "" "" ---> ""
appendString "" "Egison" ---> "Egison"
appendString "Egison" "" ---> "Egison"
appendString "Egi" "son" ---> "Egison"
```

splitString

```
splitString "," "" ---> [""]
splitString "," "2,3,5,7,11,13"
---> ["2", "3", "5", "7", "11", "13"]
```

regex

```
regex "cde" "abcdefg" ---> [("ab", "cde", "fg")]
regex "[0-9]+" "abc123defg"
---> [("abc", "123", "defg")]
regex "a*" "" ---> [("", "", "")]
```

regexCg

```
regexCg "([0-9]+),([0-9]+)" "abc,123,45,defg"
---> [("abc,", ["123", "45"], ",defg")]
```

read

```
read "3" ---> 3
read "3.14" ---> 3.14
read "[1, 2]" ---> [1, 2]
read "\"Hello world!\"" ---> "Hello world!"
```

show

```
show 3 ---> "3"
show 3.14159 ---> "3.14159"
show [1, 2] ---> "[1, 2]"
show "Hello world!" ---> "\"Hello world!\""
```

isBool

```
isBool False ---> True
```

isInteger

```
isInteger 1 ---> True
```

isRational

```
isRational 1 ---> True
isRational (1 / 2) ---> True
isRational 3.1 ---> False
```

isScalar

```
isScalar 1 ---> True
isScalar [| 1, 2 |] ---> False
```

isFloat

```
isFloat 1.0 ---> True
isFloat 1 ---> False
```

isChar

```
isChar 'c' ---> True
```

isString

```
isString "hoge" ---> True
```

isCollection

```
isCollection [] ---> True
isCollection [1] ---> True
```

isHash

```
isHash {| |} ---> True
isHash {| (1, 2) |} ---> True
```

isTensor

```
isTensor 1 ---> False
isTensor [| 1 |] ---> True
isTensor (generateTensor (+) [1, 2]) ---> True
```

2.8.2 Primitive functions for IO operations

return Takes a pure value and return an IO function that returns the value.

```
io return 1 ---> 1
```

openInputFile Takes a name of a file (string) and opens the file in read-only mode. Returns a port of the opened file.

```
let inport := openInputFile "file.txt"
```

openOutputFile Takes a name of a file (string) and opens the file in write-only (truncate) mode. Returns a port of the opened file.

```
let outport := openOuputFile "file.txt"
```

closeInputPort, closeOutputPort Takes a port and closes it.

```
closeInputPort inport
closeOutputPort outport
```

readChar Reads one character from the standard input and returns it.

```
let c := readChar ()
```

readLine Reads one line from the standard input and returns it.

```
let line := readLine ()
```

writeChar Output a given character to the standard input.

```
writeChar 'a'
```

write Output a given string to the standard input.

```
write "string"
```

readCharFromPort A variant of `readChar` that reads from the given port.

```
let c := readCharFromPort inport
```

readLineFromPort A variant of `readLine` that reads from the given port.

```
let line := readLineFromPort inport
```

writeCharToPort A variant of `writeChar` that writes to the given port.

```
writeCharToPort 'a' outport
```

writeToPort A variant of `write` that writes to the given port.

```
writeToPort "string" outport
```

isEof Returns `True` if an EOF is given in the standard input.

```
let b := isEof ()
```

flush Flushes the standard output.

```
flush ()
```

isEofPort Returns `True` if an EOF is given in the specified port.

```
let b := isEofPort inport
```

flushPort Flushes the given port.

```
flushPort outport
```

readFile Takes a name of a file (string) and returns its content as a string.

```
let lines := readFile "file.txt"
```

rand `rand n m` returns an integer in the range $[n, m]$ (including m).

f.rand Float version of `rand`.

2.9 Command-Line Options

2.9.1 -l / --load-file

Load definitions from the given file.

```
$ cat name-of-file-to-load.egi
x := 1

$ egison -l name-of-file-to-load.egi
> x
1
```

2.9.2 -t / --test

Evaluate expressions in the given file.

```
$ cat name-of-file-to-test.egi
x := 1
x + 2
"This is the third line"

$ egison -t name-of-file-to-test.egi
3
"This is the third line"
```

2.9.3 -e / --eval

Output the evaluation result of the given Egison expression.

```
$ egison -e 'matchAll [1,2,3] as list something with $x ++ _ -> x'
[[], [1], [1, 2], [1, 2, 3]]
```

2.9.4 -c / --command

Execute the given expression, which should evaluate to an IO function.

```
$ egison -c 'print (show 1)'
1
```

2.9.5 -T / --tsv

Output the evaluation result in the TSV form.

When the evaluation result is a colleciton, each of its elements is printed in a single line.

```
$ egison -T -e 'take 10 primes'
2
3
5
```

(continues on next page)

(continued from previous page)

```
7
11
13
17
19
23
29
```

When the evaluation result is a collection of collections or a collection of tuples, the elements of the inner collections are separated by a tab.

```
$ egison -T -e '[[1, 2, 3], [4, 5]]'
1       2       3
4       5
$ egison -T -e '[(1, 2, 3), (4, 5, 6)]'
1       2       3
4       5       6
```

2.9.6 `-M` / `--math`

Output the evaluation result in the specified format. The format can be chosen from `latex`, `asciimath`, `mathematica` and `maxima`.

```
$ egison -M latex
> x / y
#latex|\frac{x}{y}|#
```

2.9.7 `-S` / `--sexpr-syntax`

Use the old S-expression syntax in REPL.

```
$ egison -S
> (+ 1 2)
3
```

Note: When parsing programs in files, Egison switches the parser by the file extension. If the source file has extension `.egi`, it is interpreted in the new syntax, and if the source file has extension `.segi`, it is interpreted in the old (S-expression) syntax.

Warning: Since we are no longer taking care of the backward compatibility with the old syntax (before version 4.0.0), we recommend using the new syntax if possible.

As for Egison programs written in the old syntax, we have a tool to translate them in the new syntax. Please see *Migration Guide for the New Syntax* for details.

2.10 Migration Guide for the New Syntax

2.10.1 Automated translation with `egison-translate`

We have a tool to help users convert old Egison programs into the new syntax. `egison-translate` automatically translates Egison programs in the old S-expression syntax into the new Haskell-like syntax. It can be built from the source as follows.

```
$ git clone git@github.com:egison/egison.git
$ cd egison
$ stack init
$ stack build --fast
$ stack exec -- egison-translate name-of-file-you-want-to-convert.segi
```

Warning: `egison-translate` does not preserve comments and shebangs. Please manually add them after the translation.

2.10.2 Changes in function names

Apart from the changes in the syntax itself, some of the library/built-in variable names have also been renamed to match the new syntax. All of these changes are supported in `egison-translate`.

Changes in naming rules

The following describes the changes in the naming rule of variables. The table shows the example of such changes and the detailed description is given under the table.

	Old name	New name
1	`take-while`	`takeWhile`
2	`even?`	`isEven`
3	`member?`	`member`
4	`delete/m`	`deleteAs`

1. Names connected with hyphens `-` are converted into camelCase.
2. Unary function names that ended with a question mark `?` is prefixed with `is`.
3. Non-unary function names that ended with a question mark `foo?` now omits the last question mark.
4. Function names of the form `foo/m` are renamed as `fooAs`. For instance, `delete/m` became `deleteAs`.

Instance-wise changes

Also, some of the Lisp-inspired names have been renamed into Haskell-like names as shown in the following table.

Old name	New name
`car`	`head`
`cdr`	`tail`
`rac`	`last`
`rdc`	`init`