Boomerang
Programmer's Manual

J. Nathan Foster and Benjamin C. Pierce

with

Davi Barbosa, Aaron Bohannon, Julien Cretin, Michael Greenberg,
Alan Schmitt, Adam Magee, Alexandre Pilkiewicz, and Daniel Puller

September 2, 2009

Mailing List

Active users of Boomerang are encouraged to subscribe to the `harmony-hackers` mailing list by visiting the following URL:

`http://lists.seas.upenn.edu/mailman/listinfo/harmony-hackers`

Caveats

The Boomerang system is a work in progress. We are distributing it in hopes that others may find it useful or interesting, but it has some significant shortcomings that we know about (and, surely, some that we don't) plus a multitude of minor ones. In particular, the documentation and user interface are... minimal. Also, the Boomerang implementation has not been carefully optimized. It's fast enough to run medium-sized (thousands of lines) programs on small to medium-sized (kilobytes to tens of kilobytes) inputs, but it's not up to industrial use.

Copying

Contributing

Contributions to Boomerang—especially in the form of interesting or useful lenses—are very welcome. By sending us your code for inclusion in Boomerang, you are signalling your agreement with the license described above.

Contents

Chapter 1

Introduction

This manual describes Boomerang, a *bidirectional programming language* for ad-hoc, textual data formats. Most programs compute in a single direction, from input to output. But sometimes it is useful to take a modified *output* and "compute backwards" to obtain a correspondingly modified *input*. For example, if we have a transformation mapping a simple XML database format describing classical composers...

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <years birth="1865" death="1956"/>
    <nationality>Finnish</nationality>
  </composer>
</composers>
```

... to comma-separated lines of ASCII...

```
Jean Sibelius, 1865-1956
```

... we may want to be able to edit the ASCII output (e.g., to correct the erroneous death date above) and push the change back into the original XML. The need for *bidirectional transformations* like this one arises in many areas of computing, including in data converters and synchronizers, parsers and pretty printers, marshallers and unmarshallers, structure editors, graphical user interfaces, software model transformations, system configuration management tools, schema evolution, and databases.

1.1 Lenses

Of course, we are not interested in just any transformations that map back and forth between data—we want the two directions of the transformation to work together in some reasonable way. Boomerang programs describe a certain class of well-behaved bidirectional transformations that we call *lenses*. Mathematically, a lens l mapping between a set

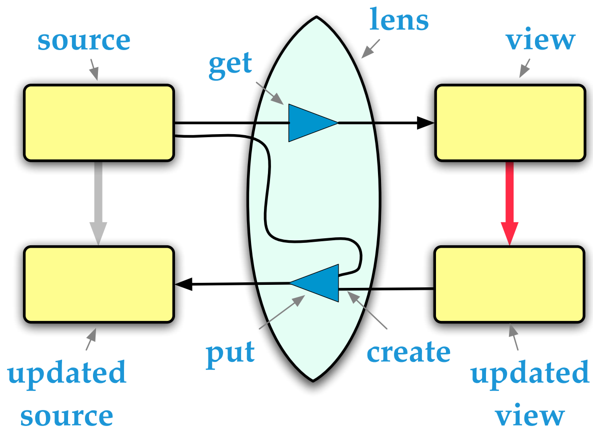

Figure 1.1: Lens Terminology

S of "source" strings and a set V of "view" ones has three components:

$$\begin{array}{rcl} l.get & \in & S \longrightarrow V \\ l.put & \in & V \longrightarrow S \longrightarrow S \\ l.create & \in & V \longrightarrow S \end{array}$$

get is the forward transformation and is a total function from S to V. The backwards transformation comes in two flavors. The first, *put*, takes two arguments, a modified V and an old S, and produces an updated S. The second, *create*, handles the special case where we need to compute a S from an V but have no S to use as the "old value". It fills in any information in S that was discarded by the *get* function (such as the nationality of each composer in the example above) with defaults. The components of a lens are shown graphically in Figure 1.1.

We say that are "well-behaved" because they obey the following "round-tripping" laws for every $s \in S$ and $v \in V$:

$$l.put\ (l.get\ s)\ s = s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ v\ s) = v \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ v) = v \qquad \text{(CreateGet)}$$

The first law requires that if *put* is invoked with an view string that is identical to the string obtained by applying *get* to the old source string—i.e., if the edit to the view string is a no-op—then it must produce the same source string. The second and third laws state that *put* and *create* must propagate all of the information in their V arguments to the S they produce. These laws capture fundamental expectations about how the components of a lens should work together.

1.2 Boomerang Overview

Boomerang is a language for writing lenses that work on strings. The key pieces of its design can be summarized as follows.

- The core of the language is a set of *string lens combinators*—primitive lenses that copying and delete strings, and ones that combine lenses using the familiar "regular operators" of union, concatenation, and Kleene-star. This core set of operators has a simple and intuitive semantics and is capable of expressing many useful transformations.

- Of course, programming with low-level combinators alone would be tedious and repetitive; we don't do this. The core combinators are embedded in a full-blown functional language with all of the usual features: let definitions, first-class functions, user-defined datatypes, polymorphism, modules, etc. This infrastructure can be used to abstract out common patterns and to build generic bidirectional libraries. We have found that they make high-level lens programming quite convenient.

- To correctly handle ordered data structures, many applications require that lenses match up corresponding pieces of the source and the view. Boomerang allows the programmer to describe these pieces (called "chunks") and how the they are aligned, choosing a method and defining the parameters for this method (such as "weights" and "threshold").

- Finally, in many applications, is often useful to be able to break the lens laws. For example, when we process XML data in Boomerang, we usually don't care whether the whitespace around elements is preserved. Boomerang includes combinators for "quotienting" lenses using "canonizers" that explicitly discard such inessential features. We call lenses that use these features *quotient lenses*.

1.3 An Example Lens

To give a sense of what programming in Boomerang is like, we will define the lens implementing the transformations between XML and CSV composers shown above.

First we define a lens `c` that handles a single `<composer>` element. It uses a number of functions defined in our XML library, as well as primitives for copying (`copy`) and deleting (`del`) strings, and for concatenating lenses (`.`).

```
let c : lens =
  Xml.elt NL2 "composer"
    begin
      Xml.simple_elt NL4 "name"
        (copy [A-Za-z ]+ . ins ", ") .
      Xml.attr2_elt_no_kids NL4 "years"
```

```
        "birth" (copy NUMBER . ins "-")
        "death" (copy NUMBER) .
      Xml.simple_elt NL4 "nationality" (del [A-Za-z]+)
   end
```

Using `c`, we then define a lens that handles a top-level `<composers>` element, enclosing a list of `<composer>`. This lens is defined using the features already described, a primitive for inserting a string (`ins`), as well as union (`|`) and Kleene star (`*`).

```
let cs : lens =
  Xml.elt NL0 "composers"
    begin
      copy EPSILON |
      c . (ins newline . c)*
    end
```

We can check that this lens actually does the transformation we want by running its *get* and *put* components on some sample data. First, let us bind the XML database to a variable (to avoid printing it many times). The `<< ... >>` is heredoc notation for a multi-line string literal.

```
let original_c : string =
<<
  <composers>
    <composer>
      <name>Jean Sibelius</name>
      <years birth="1865" death="1956"/>
      <nationality>Finnish</nationality>
    </composer>
  </composers>
>>
```

Now we test the *get* function...

```
test cs.get original_c =
<<
  Jean Sibelius, 1865-1956
>>
```

...and obtain the expected result. To check the *put* function, let us fix the error in Sibelius's death date, and *put* it back into the original XML database...

```
test cs.put
<<
Jean Sibelius, 1865-1957
>>
```

```
into original_c
=
<<

  <composers>
    <composer>
      <name>Jean Sibelius</name>
      <years birth="1865" death="1957"/>
      <nationality>Finnish</nationality>
    </composer>
  </composers>
>>
```

... again, we obtain the expected result: the new XML database reflects the change to the death date we made in the CSV string.

1.4 Getting Started

The best way to get going with Boomerang, is by working through the next "Quick Start" chapter. It contains a lightning tour of some of the main features of Boomerang the language and the system. A second step could be the section 3, which explains in details the alignment in Boomerang. After that, we suggest exploring examples (see chapter 7), and consulting the rest of this manual as needed. The chapter 6 shows how to run Boomerang, how to create your own Boomerang program and how to run a Boomerang program. Many more details can be found in our research papers on Boomerang (Bohannon et al. [2008], Foster et al. [2008]) and on lenses in general (Foster et al. [2007], Bohannon et al. [2005]), but take into account that some theorical changes have been made since these papers were published. These papers are all available from the Boomerang web page.

Good luck and have fun!

Chapter 2

Quick Start

2.1 Installation

1. Download or build the Boomerang binary:
   - Pre-compiled binaries for Linux (x86), Mac OS X (x86), and Windows (Cygwin) are available on the Boomerang webpage.
   - Alternatively, to build Boomerang from source, grab the most recent tarball and follow the instructions in INSTALL.txt
2. Add the directory containing trunk/bin to your `PATH` environment variable.
   - In Bash:
     ```
     > export PATH=$PATH:/path/to/trunk/bin
     ```
   - In Csh
     ```
     > setenv PATH $PATH:/path/to/trunk/bin
     ```

2.2 Simple Lens Programming

Now lets roll up our sleeves and write a few lenses. We will start with some very simple lenses that demonstrate how to interact with the Boomerang system. The source file we will work with is this very text file, which is literate Boomerang code. Every line in this file that begins with `#*` marks a piece of Boomerang code, and all other lines are ignored by the Boomerang interpreter.

You can run the Boomerang interpreter from the command line like this:

```
> boomerang QuickStart.src
```

You should see several lines of output beginning like this

```
Test result:
"Hello World"
Test result:
"HELLO WORLD"
...
```

Let's define the lens that was used to generate this text.

```
let l : lens = copy [A-Za-z ]+
```

This line declares a lens named 'l' using syntax based on explicitly-typed OCaml (for the functional parts, like the let declaration) and POSIX (for regular expressions). Its *get* and *put* components both copy non-empty strings of alphabetic characters or spaces.

2.2.1 Unit Tests

An easy way to interact with Boomerang is using its syntax for running unit tests (other modes of interaction, such as batch processing of files via the command line, are discussed below). For example, the following test:

```
test l.get "Hello World" = ?
```

instructs the Boomerang interpreter to calculate the result obtained by applying the *get* component of `l` to the string literal `Hello World` and print the result to the terminal (in fact, this unit test generated the output in the display above).

Example 1. Try changing the `?` above to `Hello World`. This changes the unit test from a calculation to an assertion, which silently succeeds.

Example 2. Try changing the `?` above to `HelloWorld` instead. Now the assertion fails. You should see:

```
File "./quickStart.src", line 68, characters 3-42: Unit test failed
Expected "HelloWorld" but found "Hello World"
```

When you are done with this exercise, reinsert the space to make the unit test succeed again.

Now let's examine the behavior of `l`'s *put* component.

```
test (l.put "HELLO WORLD" into "Hello World") = ?
```

You should see the following output printed to the terminal:

```
Test result:
HELLO WORLD
```

which reflects the change made to the abstract string.

2.2.2 Type Checking

The *get* and *put* components of lenses check that their arguments have the expected type. We can test this by passing an ill-typed string to `l`'s GET component:

```
test (l.get "Hello World!!") = error
```

Example 3. To see the error message that is printed by Boomerang, change the `error` above to `?` and re-run Boomerang. You should see the following message printed to the terminal:

```
File "./QuickStart.src", line 107, characters 3-35: Unit test failed
Test result: error
get built-in: run-time checking
error
c="Hello World!!" did not satisfy
((Core.matches_cex (Core.stype l)) c); counterexample: string does not match [ A-
```

Notice that Boomerang identifies a location in the string where matching failed (`HERE`). When you are done, change the `?` back to `error`.

2.3 The Composers Lens

Now let's build a larger example. We will write a lens whose GET function transforms newline-separated records of comma-separated data about classical music composers:

```
let s : string =
Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, American
Benjamin Britten, 1913-1976, English
```

into comma-separated lines where the year data is deleted:

```
let v : string =
Jean Sibelius, Finnish
Aaron Copland, American
Benjamin Britten, English
```

2.3.1 Basic Composers Lens

The lens that maps—bidirectionally—between these strings is written as follows:

```
let ALPHA : regexp = [A-Za-z ]+
let YEARS : regexp = [0-9]{4} . "-" . [0-9]{4}
let comp : lens =
```

```
          ALPHA . ", "
        . del YEARS . del ", "
        . ALPHA

let comps : lens = "" | comp . (newline . comp)*
```

We can check that `comp` works as we expect using unit tests:

```
test comps.get s = v
test comps.put v into s = s
```

There are several things to note about this program. First, we have use let-bindings to factor out repeated parts of programs, such as the regular expression named `ALPHA`. This makes programs easier to read and maintain. Second, operators like concatenation (`.`) automatically promote their arguments, according to the following subtyping relationships: `string` <: `regexp` <: `lens`. Thus, the string `", "` is automatically promoted to the (singleton) regular expression containing it, and the regular expression `ALPHA` is automatically promoted to the lens `copy ALPHA`.

Example 4. Edit the `comp` lens to abstract away the separator between fields and verify that your version has the same behavior on `c` and `a` by re-running Boomerang. Your program should look roughly like the following one:

```
let comp (sep:string) : lens = ...
let comps : lens =
  let comp_comma = comp ", " in
  ...
```

or, equivalently, one that binds `comp` to an explicit function:

```
let comp : string -> lens = (fun (sep:string) -> ... )
```

2.3.2 Resourceful Composers Lenses

The behavior of `comps` lens is not very satisfactory when the updated abstract view is obtained by changing the order of lines. For example if we swap the order of Britten and Copland, the year data from Britten gets associated to Copland, and vice versa (`<< ... >>` is Boomerang syntax for a string literal in heredoc notation.)

```
test comps.put
<<
  Jean Sibelius, Finnish
  Benjamin Britten, English
  Aaron Copland, American
>>
into
```

```
<<
  Jean Sibelius, 1865-1957, Finnish
  Aaron Copland, 1910-1990, American
  Benjamin Britten, 1913-1976, English
>>
=
<<
  Jean Sibelius, 1865-1957, Finnish
  Benjamin Britten, 1910-1990, English
  Aaron Copland, 1913-1976, American
>>
```

The root of this problem is that the PUT function of the Kleene star operator works positionally—it divides the concrete and abstract strings into lines, and invokes the PUT of comp on each pair.

Our solution is to add new combinators for specifying reorderable "chunks", assign to them an method to match up these pieces (a specie) and tune it defining weights and predicates. This is explained in details in section 3.

In our example we only need one useful case. We define a chunk using the function `dictionary` and we define a key for each chunk (`key ALPHA`). The *put* function of the following lens:

```
let ALPHA : regexp = [A-Za-z ]+
let YEARS : regexp = [0-9]{4} . "-" . [0-9]{4}
let comp : lens =
           key ALPHA . ", "
         . del YEARS . del ", "
         . ALPHA

let comps : lens = "" | <dictionary "":comp> . (newline . <dictionary "":comp>)*
```

restores lines using the name on each line as a key, rather than by position. To verify it on this example, try out this unit test:

```
test comps.put
<<
  Jean Sibelius, Finnish
  Benjamin Britten, English
  Aaron Copland, American
>>
into
<<
  Jean Sibelius, 1865-1957, Finnish
  Aaron Copland, 1910-1990, American
  Benjamin Britten, 1913-1976, English
>>
= ?
```

Note that the year data is correctly restored to each composer.

2.4 Grammars

Sometimes writing lenses using the core set of combinators is rather tedious, and we'd like a more succinct way to encode simple transformations. For example, rearranging data requires counting up individual lenses and using their positions on both sides of a transformation to form a permutation ordering list. Also, lenses don't always *look* like the transformations they encode, and one cannot easily infer what a lens is doing without running it on an example. Finally, we lack the ability to describe transformations rooted in recursive patterns using a single lens.

Our solution to these problems is to express lenses using right-recursive grammars. Each grammar is a set of named productions, each of which Boomerang compiles into a lens of the same name. Each production in turn is a set of rules, possible transformations whose union forms the definition of its corresponding lens.

A rule describes a transformation between a pair of sequenced expressions. An expression can be a lens defined in a previous grammar, a regular expression, or a string literal. Each expression present on both sides of the transformation is labeled as a variable. For example, suppose we want to write a lens `swap` that inverts a person's first and last name. Suppose we'd like it to rewrite the name "John Smith" as "Smith, John".

Without grammars, we would have to write `swap` using a permutation:

```
let FIRST : regexp = [A-Za-z]+
let LAST : regexp = [A-Za-z]+

let swap : lens =
  lens_permute #{int}[2;1;0]
               #{lens}[FIRST; ins ", " . del " "; LAST]
```

This isn't too bad, but as you can imagine, the bookkeeping gets rather difficult as the number of terms increases. Using grammars, we can more easily write the lens as:

```
let swap : lens =
  grammar
    name :: = fn:FIRST " " ln:LAST <-> ln ", " fn
  end
```

Observe that labeled terms can be reordered, and unlabeled terms are present on only one side of the transformation. To verify this lens works properly, we use the unit test:

```
test swap.get "John Smith" = "Smith, John"
```

Each production also can contain multiple rules, and each rule can be right-recursive on the entire production. We can modify the `swap` lens to write a new lens `swap_many` that operates on a semi-colon separated nonempty list of names as follows:

```
let swap_many : lens =
  grammar
    swap_many :: = fn:FIRST " " ln:LAST <-> ln ", " fn
                 | fn:FIRST " " ln:LAST "; " ns:swap_many
                     <-> ln ", " fn "; " ns
  end
```

Here, the first rule for `swap_many` is precisely the same as the rule for `swap` and behaves the same way: it inverts the order of a single name. The second rule is a bit more interesting. It inverts the order of a single name and concatenates the result with another application of the production. The production will ultimately have to use the first rule to terminate, since the second rule always insists on an additional application of the production. We can test it on a list of two names:

```
test swap_many.get "John Smith; Jane Doe" = "Smith, John; Doe, Jane"
```

Finally, we can rely on the previously defined lens `swap` in order to write `swap_many` more cleanly as follows:

```
let swap_many' =
  grammar
    swap_many ::= n:swap <-> n
                | n:swap "; " ns:swap_many
                    <-> n "; " ns
  end
```

and test that it behaves just as before:

```
test swap_many'.get "John Smith; Jane Doe" = "Smith, John; Doe, Jane"
```

Grammars are fully-integrated within the Boomerang system, and as such the resulting lenses produced behave just as an other well-formed lenses. The `swap` lens can be used as part of the definition of a subsequent lens `condense` that removes extraneous personal information:

```
let AGE : regexp = [0-9]+
let GENDER : regexp = "M" | "F"

let condense : lens =
  swap . del ", " . del AGE . del ", " . del GENDER
```

and verify the correct behavior with a couple of unit tests:

```
test condense.get "John Smith, 24, M" = "Smith, John"
test condense.put "Hancock, John" into "John Smith, 24, M"
  = "John Hancock, 24, M"
```

Taking this one step further, the lens `condense` also can be used in a subsequent grammar `pair` that takes a list of two newline-separated individuals and pairs them up:

```
let pair : lens =
  grammar
    pair ::= c1:condense newline c2:condense
               <-> "(" c1 " & " c2 ")"
  end
```

which in turn can be used to define the lens `pair_many`, which operates on a list with an even number of names and pairs them up:

```
let pair_many : lens =
  grammar
    pair_many ::= p:pair <-> p
                | p:pair newline ps:pair_many
                    <-> p newline ps
  end
```

and verify correct behavior:

```
let two_names : string =
<<
John Smith, 24, M
Jane Doe, 23, F
>>

test pair.get two_names = "(Smith, John & Doe, Jane)"

let many_names : string =
<<
John Smith, 24, M
Jane Doe, 23, F
Brad Pitt, 45, M
Angelina Jolie, 33, F
>>

test pair_many.get many_names =
<<
(Smith, John & Doe, Jane)
(Pitt, Brad & Jolie, Angelina)
>>
```

Finally, we can take the names from the output and easily rearrange them to present how their names would be displayed as a married couple (assuming the last name that appears first is used as their married name):

```
let marry : lens =
  grammar
    marry ::= "(" ln1:LAST ", " fn1:FIRST " & " LAST ", " fn2:FIRST ")"
                <-> fn1 " and " fn2 " " ln1
  end
```

and test it by composing the get function of `marry` and `pair`:

```
test marry.get (pair.get two_names) = "John and Jane Smith"
```

Notice that the last name of the second person in the pair isn't labeled in the grammar, since it isn't copied over to the output.

2.4.1 Rewriting the Composers Lens with Grammars

Using right-recursive grammars, we can rewrite the basic composers lenses as follows:

```
let comp : lens =
  grammar
    comp ::= nm:(key ALPHA) ", " YEARS ", " cntry:ALPHA
               <-> nm ", " cntry
  end
let comps : lens =
  grammar
    comps ::= c:<dictionary "":comp> <-> c
           |  c:<dictionary "":comp> newline cs:comps <-> c newline cs
  end
```

and verify it with the same unit tests as earlier:

```
test comps.get s = v
test comps.put v into s = s
test comps.put <<
Jean Sibelius, Finnish
Aaron Copland, Yankee
>> into s = <<
Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, Yankee
>>
```

2.4.2 Mutually-Recursive Grammars

Boomerang also supports grammars with mutually-recursive productions, as in the following example:

```
let (pos,neg) : lens * lens =
  grammar
    pos ::= "positive" <-> "+"
          | "positive " n:neg <-> "+ " n
    and neg ::= "negative" <-> "-"
              | "negative " p:pos <-> "- " p
  end

test pos.get "positive" = "+"
test pos.get "positive negative positive negative" = "+ - + -"
test neg.create "- + -" = "negative positive negative"
```

The behavior of these lenses is as follows:


```
test pos.get "positive" = "+"
test pos.get "positive negative positive negative" = "+ - + -"
test neg.create "- + -" = "negative positive negative"
```

Chapter 3

Alignment

When updating a source, usually we want the lens to be able to match up pieces of the updated view with the corresponding pieces of the old view, to restore the hidden information. We already saw this problem in the composers lens of the Quick Start, when swapping the order of composers the year data is not swapped by the PUT function. We solved this using a dictionary and a key functions defined in `Core.boom`.

In this section we explain in more details how this works.

3.1 Tags

Tag is a type defined in `Core.boom`. It is a quadruple with a *specie*, a *predicate*, a default *key annotation* and a string. The first one define which method Boomerang will use to align the chunks. The second is a way for the programmer to forbid a match between two chunks, and for the moment, it comes in only one flavor: a threshold. The third one is explained in the following section. Finally, the last one is a identifier[1].

The identifier is used to specify groups of chunks that are aligned independently (Boomerang will match only chunks with the same identifier). During one alignment, chunks with the same identifier should also agree on the tag, i.e., the specie and the predicate should be the same.

Briefly, the species are

Positional Chunks are aligned positionally.

Diffy Non crossing alignment minimizing the cost.

Greedy A greedy algorithm to find an alignment with a low cost. In each step it match the first pair of chunks with the smallest cost.

Setlike An alignment minimizing the total cost.

[1]this is actually what was called "tag" in the old versions of Boomerang

where cost is the sum of string distances between the matched keys (plus the cost of nested alignment, if any). We will see them in more details as we show some examples.

A threshold x forbid any match between chunks that does not conserve at least $x\%$ of the key. For example, a threshold of 0 allows all alignments and a threshold of 100 allows only chunks with exactly the same key to be matched.

3.2 Keys

Key annotations are used to indicate which part of the chunk is relevant and should be used for the alignment. The parts of the chunk annotated with key are used for the alignment and the parts annotated with nokey are not used. These annotations can be placed using the functions `key` and `nokey`, and these functions does not override previous definitions (a `key` inside a `nokey` will be used for the alignment). We give two more functions, `force_key` and `force_nokey` to override previous definitions.

When we have nested chunks, the keys in a nested chunk are not used for the alignment of an enclosing chunk. The functions to set key annotations does not go inside other chunks.

If a part of a chunk does not have a key annotation, it will use the default value given by the chunk (defined with the tag).

3.3 Learning with examples

We will use the same composers example from the Quick Start:

```
let ALPHA : regexp = [A-Za-z ]+
let YEARS : regexp = [0-9]{4} . "-" . [0-9]{4}
let comp : lens =
           key ALPHA . ", "
         . del YEARS . del ", "
         . nokey ALPHA

let create_comps (chunk:lens) : lens = "" | chunk . (newline . chunk)*
```

3.3.1 Dictionary

Dictionary lenses can be written using the *greedy* alignment (see next section), for this, we only need to use the function `dictionary` that generates a `Tag`. Using dictionary, Boomerang only matches two chunks when the key is exactly the same.

```
let comps : lens = create_comps <dictionary "":comp>
```

```
test comps.put
<<
  Benjamin Briten, English
  Benjamin Britten, Yankee
  Aron Copland, American
>>
into
<<
  Jean Sibelius, 1865-1957, Finnish
  Aaron Copland, 1910-1990, American
  Benjamin Britten, 1913-1976, English
>>
=
<<
  Benjamin Briten, 0000-0000, English
  Benjamin Britten, 1913-1976, Yankee
  Aron Copland, 0000-0000, American
>>
```

If more then one chunk has the same key, the dictionary will match then consecutively in the same order as they are in the view and in the source. For example:

```
test comps.put
<<
  Repeated Key, English
  Repeated Key, American
  Repeated Key, Finnish
>>
into
<<
  Repeated Key, 1111-1111, Finnish
  Repeated Key, 2222-2222, American
  Benjamin Britten, 1913-1976, English
>>
=
<<
  Repeated Key, 1111-1111, English
  Repeated Key, 2222-2222, American
  Repeated Key, 0000-0000, Finnish
>>
```

It starts matching the first entry in the view (`Repeated Key, English`) with the first entry in the source with the same key (`Repeated Key, 1111-1111, Finnish`), and go on[2]. For the last entry in the view, there is no more unmatched key `Repeated Key` in the source, so a create is used.

[2]the real algorithm (greedy alignment) is different, but has the same result

3.3.2 Greedy

The greedy constructs the alignment iteratively, adding at each step the first put with the smallest cost, i.e., choosing at each step the best pair.

```
let comps : lens = create_comps <greedy 0 "":comp>

test comps.put
<<
  Benjamin Briten, English
  Benjamin Brtten, Yankee
  Aron Copland, American
>>
into
<<
  Jean Sibelius, 1865-1957, Finnish
  Aaron Copland, 1910-1990, American
  Benjamin Britten, 1913-1976, English
>>
=
<<
  Benjamin Briten, 1913-1976, English
  Benjamin Brtten, 1865-1957, Yankee
  Aron Copland, 1910-1990, American
>>
```

in this example, the string distance between the first two key in the updated view and the old key of Benjamin Britten are the same, so the algorithm chose the first one for the put. At the end, even if Benjamin Brtten and Jean Sibelius are very different, it will chose this pair for a put (because they are the only remaining chunks). The function of the threshold is to forbidden this kind of behavior.

```
test comps.put
<<
  Benjamin, English
>>
into
<<
  Benjamin Britten, 1913-1976, English
  Benjamin Cooke, 1734-1793, English
>>
=
<<
  Benjamin, 1734-1793, English
>>
```

However, introducing thresholds we have:

```
let comps : lens = create_comps <greedy 50 "":comp>

test comps.put
<<
  Ben, English
  Sibelius, Finnish
>>
into
<<
  Benjamin Britten, 1913-1976, English
  Benjamin Cooke, 1734-1793, English
  Jean Sibelius, 1865-1957, Finnish
>>
=
<<
  Ben, 0000-0000, English
  Sibelius, 1865-1957, Finnish
>>
```

3.3.3 Setlike

The greedy tries to minimize the alignment in a very naive way. On the other hand, the setlike really minimizes the total cost of the alignment. However the algorithm is more complex and slower. Even if it is deterministic, it is not predictable which will be the answer when more then one answer is correct.

```
let comps : lens = create_comps <setlike 50 "":comp>

test comps.put
<<
  Benjamin Briten, English
  Benjamin Brtten, Yankee
  Aron Copland, American
>>
into
<<
  Jean Sibelius, 1865-1957, Finnish
  Aaron Copland, 1910-1990, American
  Benjamin Britten, 1913-1976, English
>>
=
<<
  Benjamin Briten, 0000-0000, English
```

```
  Benjamin Brtten, 1913-1976, Yankee
  Aron Copland, 1910-1990, American
>>
```

in the previous example, another answer were possible:

```
<<
  Benjamin Briten, 1913-1967, English
  Benjamin Brtten, 0000-0000, Yankee
  Aron Copland, 1910-1990, American
>>
```

Let's see the difference between greedy and setlike with an example:

```
let comps : lens = create_comps <greedy 0 "":comp>

test comps.put
<<
  abd, first
  acdefg, second
>>
into
<<
  xyzabd, 1111-1111, something
  acd, 2222-2222, something
>>
=
<<
  abd, 2222-2222, first
  acdefg, 1111-1111, second
>>
```

while the setlike really minimizes the alignment:

```
let comps : lens = create_comps <setlike 0 "":comp>

test comps.put
<<
  abd, first
  acdefg, second
>>
into
<<
  xyzabd, 1111-1111, something
  acd, 2222-2222, something
>>
=
```

```
<<
  abd, 1111-1111, first
  acdefg, 2222-2222, second
>>
```

3.3.4 Positional

The positional just align the chunks sequentially. It is almost the same as does not have chunks, but it allows some advanced techniques that are not possible without chunks. For example, it is possible to pass an information across an union:

```
let notacross_union =
  "L" . del [a-z]* | "R" . copy [a-z]*
test notacross_union.put "L" into "Rfromr" = "L"

let across_union =
  "L" . <positional "":del [a-z]*> | "R" . <positional "":copy [a-z]*>
test across_union.put "L" into "Rfromr" = "Lfromr"
```

Chapter 4

The Boomerang Language

The Boomerang language provides convenient concrete syntax for writing lenses (and strings, regular expressions, canonizers, etc.). The concrete syntax is based on an explicitly-typed core fragment of OCaml. It includes user-defined datatypes and functions, modules, unit tests, and special syntax for constructing regular expressions and for accessing the components of lenses.

4.1 Lexing

Space, newline and tab characters are whitespace. Comments are equivalent to whitespace and are delimited by `(*` and `*)`; comments may be nested.

4.1.1 String Literals

String literals can be any sequence of characters and escape sequences enclosed in double-quotes. The escape sequences `\"`, `\\`, `\b`, `\n`, `\r`, and `\t` stand for the characters double-quote, backslash, backspace, newline, vertical tab, and tab. To facilitate lining up columns in indented string literals, within a string, a newline followed by whitespace and then `|` is equivalent to a single newline. For example,

```
  "University
  |of
  |Pennsylvania"
```

is equivalent to both

```
"University
of
Pennsylvania"
```

(in the leftmost column) and

```
"University\nof\nPennsylvania"
```

(anywhere). String literals can also be specified using "here document" (heredoc) notation, delimited by << and >>. If the initial << is followed by a newline and sequence of space characters, that indentation is used for the rest of the block. For example, the following string

```
<<
  University
  of
  Pennsylvania
>>
```

is equivalent to the previous ones.

4.1.2 Identifiers

Ordinary identifiers are non-empty strings drawn from the following set of characters

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
' _ - @
```

The first symbol of an identifier must be a non-numeric character. The following keywords

module	open	let	in	fun
begin	end	test	match	with
type	error	char	string	regexp
lens	int	bool	canonizer	unit
of	into	where	forall	lt
leq	gt	geq	true	false
cex	grammar	and		
.get	.put	.create	.canonize	.choose
.stype	.astype	.domain_type	.vtype	.avtype
.codomain_type	.bij	aregexp	skeleton_set	resource_set

and symbols

```
( ) ; . & * - _ + ! -> => <=> <-> =
{ } # [ ] < > , : ^ ~ / ? \
```

are reserved.

Some of the parsing rules distinguish several different kinds of identifiers. The lexer produces different tokes for uppercase (*UIdent*) and lowercase (*LIdent*) identifiers. Additionally, the lexer produces special tokens for qualified identifiers (*QualIdent*), which have the form `M.N.x`, and type variable identifiers (*TyVarIdent*), which have the form `'a`.

4.1.3 Regular Expressions

Character classes are specified within `[` and `]` using POSIX notation. The `^` character indicates a negated character class. For example, `[A-Z]` is the set of upper case characters, `[0-9]` the set of decimal digits, `[^]` the full set of ASCII characters, and `[^\n\t ]` the set of non-newline, non-tab, non-space characters.

4.2 Parsing

This section gives a formal definition of Boomerang syntax as an EBNF grammar. The productions for each syntactic category are followed by a brief explanation. In grammar rules we adopt the following conventions:

- Literals are written in a typewriter font and enclosed in quotes: e.g., '`module`';
- Non-terminals and tokens are enclosed in angle brackets: e.g., ⟨*Exp*⟩;
- Optional elements are enclosed in square brackets: e.g., ['`:`' ⟨*Sort*⟩];
- Terms are grouped using parentheses;
- Optional and repeated terms are specified using ? (optional), * (0 or more), and + (1 or more).

4.2.1 Modules and Declarations

⟨*CompilationUnit*⟩ ::= '`module`' ⟨*UIdent*⟩ '=' ('`open`' ⟨*Qid*⟩)* ⟨*Decl*⟩*

⟨*Decl*⟩ ::= '`module`' ⟨*LIdent*⟩ '=' ⟨*Decl*⟩* '`end`'
 | '`type`' ⟨*TyVarList*⟩ ⟨*LIdent*⟩ '=' ⟨*DTSortList*⟩
 | '`let`' ⟨*Id*⟩ (⟨*Param*⟩)+ ['`:`' ⟨*Sort*⟩] '=' ⟨*Exp*⟩
 | '`let`' ⟨*LetPat*⟩ (⟨*Param*⟩)+ ['`:`' ⟨*Sort*⟩] '=' ⟨*Exp*⟩
 | '`test`' ⟨*InfixExp*⟩ '=' ⟨*TestResExp*⟩
 | '`test`' ⟨*InfixExp*⟩ '`:`' ⟨*TestResSort*⟩

A Boomerang compilation unit contains a single module declaration, such as `module Foo`, which must appear in a file named `foo.src` (for "literate" sources) or `foo.boom` (for plain sources). Boomerang modules are only used to group declarations into a common namespace (in particular, Boomerang does not support module signatures or sealing). A module consists of a sequence of `open` declarations, which import all the declarations from another module into the namespace, followed by a sequence of declarations. A declaration is either a nested `module`, a `type`, a `let`, or a unit test.

Unit Tests

Boomerang supports inline unit tests, which are executed when the system is run in testing mode (see Section 6.1).

⟨*TestResExp*⟩ ::= '`?`'
 | '`error`'
 | ⟨*AppExp*⟩

⟨*TestRestSort*⟩ ::= '`?`'
 | ⟨*Sort*⟩

Unit tests have one of the following forms:

```
test (copy [A-Z]*).get "ABC" = "ABC"
test (copy [A-Z]*).get "ABC" = ?
test (copy [A-Z]*).get "123" = error
test (copy [A-Z]*).get "ABC" : string
test (copy [A-Z]*).get "ABC" : ?
```

The first form, `test` e_1 = e_2, checks that e_1 and e_2 evaluate to identical values. The two expressions must have compatible sorts with a defined equality operation. We often use this kind of test to print and check the behavior of the *get*, *put*, and *create* components of lenses. A unit test of the form `test` e `= ?` evaluates e and prints the result. The third form of unit test, `test` e `= error`, checks that an exception is raised during evaluation of e. This kind of test is used to check that a lens correctly checks the side conditions on its inputs. Notice that only evaluation errors are tested. Finally, unit tests of the form `test` e `:` s and `test` e `: ?` test the sort of e rather than its value.

4.2.2 Expressions

⟨*Exp*⟩ ::= '`let`' ⟨*Id*⟩ (⟨*Param*⟩)+ ['`:`' ⟨*Sort*⟩] '`=`' ⟨*Exp*⟩ '`in`' ⟨*Exp*⟩
 | '`let`' ⟨*LetPat*⟩ ['`:`' ⟨*Sort*⟩] '`=`' ⟨*Exp*⟩ '`in`' ⟨*Exp*⟩
 | ⟨*FunExp*⟩

⟨*FunExp*⟩ ::= '`fun`' (⟨*Param*⟩)+ ['`:`' ⟨*Sort*⟩] '`->`' ⟨*Exp*⟩
 | ⟨*CExp*⟩ ['`$`' ⟨*FunExp*⟩] ⟨*CaseExp*⟩ ::= '`match`' ⟨*ComposeExp*⟩ '`with`' ⟨*BranchList*⟩ ['`:`' ⟨*Sort*⟩]
 | ⟨*ComposeExp*⟩

⟨*ComposeExp*⟩ ::= ⟨*ComposeExp*⟩ '`;`' ⟨*CommaExp*⟩
 | ⟨*CommaExp*⟩

⟨*CommaExp*⟩ ::= ⟨*CommaExp*⟩ '`,`' ⟨*BarExp*⟩
 | ⟨*BarExp*⟩

⟨*BarExp*⟩ ::= ⟨*OBarExp*⟩
 | ⟨*DBarExp*⟩
 | ⟨*EqualExp*⟩

⟨*OBarExp*⟩ ::= ⟨*OBarExp*⟩ '|' ⟨*EqualExp*⟩
 | ⟨*EqualExp*⟩ '|' ⟨*EqualExp*⟩

⟨*DBarExp*⟩ ::= ⟨*DBarExp*⟩ '||' ⟨*EqualExp*⟩
 | ⟨*EqualExp*⟩ '||' ⟨*EqualExp*⟩

⟨*EqualExp*⟩ ::= ⟨*AppExp*⟩ '=' ⟨*AppExp*⟩
 | ⟨*InfixExp*⟩

⟨*InfixExp*⟩ ::= ⟨*DotExp*⟩
 | ⟨*TildeExp*⟩
 | ⟨*AmpExp*⟩
 | ⟨*AmpAmpExp*⟩
 | ⟨*RewriteExp*⟩
 | ⟨*LensComponentExp*⟩
 | [⟨*InfixExp*⟩] '−' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`lt`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`leq`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`gt`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`geq`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩

⟨*DotExp*⟩ ::= ⟨*DotExp*⟩ '`.`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`.`' ⟨*AppExp*⟩

⟨*TildeExp*⟩ ::= ⟨*TildeExp*⟩ '`~`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`~`' ⟨*AppExp*⟩

⟨*AmpExp*⟩ ::= ⟨*AmpExp*⟩ '`&`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`&`' ⟨*AppExp*⟩

⟨*AmpAmpExp*⟩ ::= ⟨*AppExp*⟩ '`&`' ⟨*AmpAmpExp*⟩
 | ⟨*AppExp*⟩ '`&`' ⟨*AppExp*⟩

⟨*RewriteExp*⟩ ::= ⟨*AppExp*⟩ '`\ltsym-\btsym`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`\ltsym=\btsym`' ⟨*AppExp*⟩

⟨*LensComponentExp*⟩ ::= ⟨*AppExp*⟩ '`.get`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`.put`' ⟨*AppExp*⟩ '`into`' ⟨*AppExp*⟩
 | ⟨*AppExp*⟩ '`.create`' ⟨*AppExp*⟩

| ⟨*AppExp*⟩ ‘`.canonize`’ ⟨*AppExp*⟩
| ⟨*AppExp*⟩ ‘`.choose`’ ⟨*AppExp*⟩

⟨*AppExp*⟩ ::= ⟨*AppExp*⟩ ⟨*RepExp*⟩
| ⟨*RepExp*⟩

⟨*RepExp*⟩ ::= ⟨*TyExp*⟩ ⟨*Rep*⟩
| ⟨*TyExp*⟩

⟨*TyExp*⟩ ::= ⟨*TyExp*⟩ ‘`{`’ ⟨*Sort*⟩ ‘`}`’
| ⟨*AExp*⟩

⟨*AExp*⟩ ::= ‘`(`’ ⟨*Exp*⟩ ‘`)`’
| ‘`begin`’ ⟨*Exp*⟩ ‘`end`’
| ⟨*Qid*⟩
| ⟨*MatchExp*⟩
| ‘`#`’ ‘`{`’ ⟨*SortList*⟩ ‘`}`’ ⟨*List*⟩
| ⟨*Character*⟩
| ⟨*Integer*⟩
| ⟨*Boolean*⟩
| ⟨*CharSet*⟩
| ⟨*NegCharSet*⟩
| ⟨*String*⟩
| ⟨*RegExpString*⟩
| ‘`()`’
| ‘`grammar`’ ⟨*Productions*⟩ ‘`end`’
| ⟨*AExp*⟩ ‘`.stype`’
| ⟨*AExp*⟩ ‘`.vtype`’
| ⟨*AExp*⟩ ‘`.astype`’
| ⟨*AExp*⟩ ‘`.avtype`’
| ⟨*AExp*⟩ ‘`.domain_type`’
| ⟨*AExp*⟩ ‘`.codomain_type`’
| ⟨*AExp*⟩ ‘`.bij`’

⟨*MatchExp*⟩ ::= <⟨*Exp*⟩ >
| <⟨*AppExp*⟩ ‘`:`’ ⟨*Exp*⟩ >

Branches

⟨*Branch*⟩ ::= ⟨*Pat*⟩ ‘`->`’ ⟨*EqualExp*⟩

⟨*BranchList*⟩ ::= [‘|’] ⟨*Branch*⟩ (‘|’ ⟨*Branch*⟩)*

Repetitions

⟨*Rep*⟩ ::= '*'
| '+'
| '?'
| '{' ⟨*Integer*⟩ '}'
| '{' ⟨*Integer*⟩ ',' ⟨*Integer*⟩ '}'

Lists

⟨*List*⟩ ::= '[]'
| '[' ⟨*CommaExp*⟩ (';' ⟨*CommaExp*⟩)* ']'

Grammars

⟨*Atom*⟩ ::= ⟨*AExp*⟩
| ⟨*LIdent*⟩ ':' ⟨*Qid*⟩

⟨*Atoms*⟩ ::= ⟨*Atom*⟩+

⟨*Aexps*⟩ ::= ⟨*Aexp*⟩+

⟨*Rule*⟩ ::= ⟨*Atoms*⟩ <->⟨*Aexp*⟩

⟨*Rules*⟩ ::= ['|'] ⟨*Rule*⟩ ('|' ⟨*Rule*⟩)*

⟨*Production*⟩ ::= ⟨*LIdent*⟩ '::=' ⟨*Rules*⟩

⟨*Productions*⟩ ::= ⟨*Production*⟩ ('and' ⟨*Production*⟩)*

4.2.3 Identifiers

⟨*Id*⟩ ::= ⟨*LIdent*⟩
| ⟨*UIdent*⟩

⟨*QId*⟩ ::= ⟨*LIdent*⟩
| ⟨*UIdent*⟩
| ⟨*QualIdent*⟩

⟨*QVar*⟩ ::= ⟨*LIdent*⟩
| ⟨*QualIdent*⟩

4.2.4 Parameters

⟨*Param*⟩ ::= '(' ⟨*Id*⟩ ':' ⟨*Sort*⟩ ')'
| '(' ⟨*LIdent*⟩ ':' ⟨*Sort*⟩ 'where' ')'
| '(' ⟨*LIdent*⟩ ': lens in ? <->' ⟨*AppExp*⟩ ')'
| '(' ⟨*LIdent*⟩ ': lens in ? <=>' ⟨*AppExp*⟩ ')'
| '(' ⟨*LIdent*⟩ ': lens in' ⟨*AppExp*⟩ '<->?)'
| '(' ⟨*LIdent*⟩ ': lens in' ⟨*AppExp*⟩ '<=>?)'
| '(' ⟨*LIdent*⟩ ': lens in' ⟨*AppExp*⟩ '<->' ⟨*AppExp*⟩ ')'
| '(' ⟨*LIdent*⟩ ': lens in' ⟨*AppExp*⟩ '<=>' ⟨*AppExp*⟩ ')'
| '(' ⟨*LIdent*⟩ ': string in' ⟨*Exp*⟩ ')'
| '(' ⟨*TyVarIdent*⟩ ')'
| ⟨*TyVarIdent*⟩
| '()'

4.2.5 Sorts

⟨*Sort*⟩ ::= 'forall' ⟨*TyVarIdent*⟩ '=>' ⟨*Sort*⟩
| ⟨*ArrowSort*⟩

⟨*ArrowSort*⟩ ::= ⟨*ProductSort*⟩ '->' ⟨*ArrowSort*⟩
| '(' ⟨*LIdent*⟩ ':' ⟨*ProductSort*⟩ '->' ⟨*ArrowSort*⟩ ')'
| ⟨*ProductSort*⟩

⟨*ProductSort*⟩ ::= ⟨*ProductSort*⟩ '->' ⟨*DataTypeSort*⟩
| ⟨*DataTypeSort*⟩

⟨*DataTypeSort*⟩ ::= ⟨*BSort*⟩ [⟨*QVar*⟩]
| '(' ⟨*Sort*⟩ ',' ⟨*SortList*⟩ ')' ⟨*QVar*⟩

⟨*BSort*⟩ ::= '(' ⟨*Sort*⟩ ')'
| '(' ⟨*Sort*⟩ 'where' ⟨*Exp*⟩ ')'
| '(' ⟨*LIdent*⟩ ':' ⟨*Sort*⟩ 'where' ')'
| '(lens in ? <->' ⟨*AppExp*⟩ ')'
| '(lens in ? <=>' ⟨*AppExp*⟩ ')'
| '(lens in' ⟨*AppExp*⟩ '<->?)'
| '(lens in' ⟨*AppExp*⟩ '<=>?)'
| '(lens in' ⟨*AppExp*⟩ '<->' ⟨*AppExp*⟩ ')'
| '(lens in' ⟨*AppExp*⟩ '<=>' ⟨*AppExp*⟩ ')'
| '(string in' ⟨*Exp*⟩ ')'
| ⟨*ASort*⟩

⟨*ASort*⟩ ::= ⟨*QVar*⟩
| 'char'

| ‘string’
| ‘regexp’
| ‘aregexp’
| ‘skeleton_set’
| ‘resource_set’
| ‘lens’
| ‘int’
| ‘bool’
| ‘canonizer’
| ‘unit’
| ⟨*TyVar*⟩

⟨*TyVar*⟩ ::= ⟨*TyVarIdent*⟩

⟨*TyVarList*⟩ ::= ⟨*TyVar*⟩
| ‘(’ ⟨*TyVar*⟩ (‘,’ ⟨*TyVar*⟩)* ‘)’

⟨*DTSort*⟩ ::= ⟨*UIdent*⟩
| ⟨*UIdent*⟩ ‘of’ ⟨*Sort*⟩

⟨*DTSortList*⟩ ::= ⟨*DTSort*⟩ (‘|’ ⟨*DTSort*⟩)*

4.2.6 Patterns

⟨*Pat*⟩ ::= ⟨*Pat*⟩ ‘,’ ⟨*ListPat*⟩
| ⟨*Pat*⟩ ‘,’ ⟨*ConPat*⟩
| ⟨*ListPat*⟩
| ⟨*ConPat*⟩

⟨*LetPat*⟩ ::= ⟨*LetPat*⟩ ‘,’ ⟨*ListPat*⟩
| ⟨*LetPat*⟩ ‘,’ ⟨*ConPat*⟩
| ⟨*QualIdent*⟩ ⟨*APat*⟩
| ⟨*APat*⟩

⟨*ConPat*⟩ ::= ⟨*UIdent*⟩ ⟨*APat*⟩
| ⟨*QualIdent*⟩ ⟨*APat*⟩
| ⟨*APat*⟩

⟨*APat*⟩ ::= ‘_’
| ⟨*LIdent*⟩
| ‘()’
| ⟨*Integer*⟩
| ⟨*Boolean*⟩
| ‘cex’ ⟨*Pat*⟩

| ⟨*String*⟩
| ⟨*UIdent*⟩
| ⟨*Qualident*⟩
| '(' ⟨*Pat*⟩ ')' ⟨*ListPat*⟩ ::= '[]'
| ⟨*ConPat*⟩ '::' '_'
| ⟨*ConPat*⟩ '::' ⟨*LIdent*⟩
| ⟨*ConPat*⟩ '::' ⟨*ListPat*⟩

4.3 Coercions

Some coercions are automatically inserted by the type checker on programs that use subtyping. These coercions are

- `string` to `regexp` (can be made manually using `str`)
- `regexp` to `aregexp` (can be made manually using `rxlift`)
- `regexp` to `lens` (can be made manually using `copy`)

4.4 Operators

To make it simple to write lenses and understand them, Boomerang has some operators that desugars into functions defined in Boomerang libraries.

In the table 4.2, the notation `[type]` should be replaced by the type of `expr`, for example, `[a-z]?` desugars to `regexp_iter [a-z] 0 1`.

<table>
<tr><th>operator</th><th>applied to</th><th>resolves to</th></tr>
<tr><td>=</td><td>anything (polymorphic)</td><td>equals{type}</td></tr>
<tr><td>gt</td><td>int</td><td>bgt</td></tr>
<tr><td>lt</td><td>int</td><td>blt</td></tr>
<tr><td>geq</td><td>int</td><td>bgeq</td></tr>
<tr><td>leq</td><td>int</td><td>bleq</td></tr>
<tr><td>&&</td><td>bool</td><td>land</td></tr>
<tr><td>||</td><td>bool</td><td>lor</td></tr>
<tr><td>||</td><td>lens</td><td>union</td></tr>
<tr><td>|</td><td>regexp</td><td>union</td></tr>
<tr><td>|</td><td>lens</td><td>disjoint_union</td></tr>
<tr><td>&</td><td>regexp</td><td>inter</td></tr>
<tr><td>-</td><td>regexp</td><td>diff</td></tr>
<tr><td>-</td><td>int</td><td>minus</td></tr>
<tr><td>.</td><td>string</td><td>string_concat</td></tr>
<tr><td>.</td><td>regexp</td><td>regexp_concat</td></tr>
<tr><td>.</td><td>aregexp</td><td>aregexp_concat</td></tr>
<tr><td>.</td><td>lens</td><td>lens_concat</td></tr>
<tr><td>.</td><td>canonizer</td><td>canonizer_concat</td></tr>
<tr><td>~</td><td>lens</td><td>lens_swap</td></tr>
<tr><td><-></td><td>lens</td><td>set</td></tr>
<tr><td><=></td><td>lens</td><td>rewrite</td></tr>
</table>

Table 4.1: Infix operators

operator	applied to	resolves to
expr+	regexp, aregexp, canonizer	[type]_iter expr 1 (-1)
expr+	lens	lens_plus
expr*	regexp, aregexp, canonizer	[type]_iter expr 0 (-1)
expr*	lens	lens_star
expr?	regexp, aregexp, canonizer	[type]_iter expr 0 1
expr?	lens	lens_option
expr{n,m}	regexp, aregexp, lens, canonizer	[type]_iter expr n m
expr{n,}	regexp, aregexp, lens, canonizer	[type]_iter expr n (-1)

Table 4.2: Postfix operators

notation	resolves to
`lens.bij`	`bij`
`lens.get`	`get`
`lens.put`	`put`
`lens.create`	`create`
`lens.stype`	`stype`
`lens.domain_type`	`stype`
`lens.astype`	`astype`
`lens.vtype`	`vtype`
`lens.codomain_type`	`vtype`
`lens.avtype`	`avtype`

Table 4.3: Lens record-style projection notation

notation	desugars to
`<aregexp>`	`aregexp_match (greedy 0 "")`
`<lens>`	`lens_match (greedy 0 "")`
`<tag:aregexp>`	`aregexp_match`
`<tag:lens>`	`lens_match`
`lens in S <-> V`	`in_lens_type`
`lens in S <=> V`	`in_bij_lens_type`

Table 4.4: Other notations

Chapter 5

The Boomerang Libraries

The Boomerang system includes an assortment of useful primitive lenses, regular expressions, canonizers, as well as derived forms. All these are described in this chapter, grouped by module.

In most cases, the easiest way to understand what a lens does is to see it in action on examples; most lens descriptions therefore include several unit tests, using the notation explained in Section 4.2.1.

More thorough descriptions of most of the primitive lenses can be found in our technical papers Bohannon et al. [2008], Foster et al. [2008]. The long versions of those papers include proofs that all of our primititives are "well behaved,". However, for getting up to speed with Boomerang programming, the shorter (conference) versions should suffice.

5.1 The Core Definitions

The first module, `Core`, imports primitive values (defined in the host language, OCaml) to Boomerang. In `Core`, we do not use any overloaded or infix operators (e.g., `.`, `|`, `~`, `-`, `*`) because the Boomerang type checker resolves these symbols to applications of functions defined in `Core`. (We do this because it facilitates checking the preconditions on primitive values using dependent refinement types.)

Values defined in `Core` are available by default in every Boomerang program.

5.1.1 Equality

`equals` The polymorphic `equals` operator is partial: comparing function, lens, or canonizer values raises a run-time exception. The infix = operator desugars into `equals`, instantiated with appropriate type arguments.

```
let equals : forall 'a => 'a -> 'a -> bool

test equals{string} "ABC" "ABC" = true
```

```
test equals{string} "ABC" "123" = false
test equals{char} 'A' '\065' = true
test equals{string -> string}
  (fun (x:string) -> x) (fun (y:string) -> y) = error
```

5.1.2 Booleans

`land, lor, not, implies` These operators are the standard functions on booleans. The infix operators `&&` and `||` resolve to `land` and `lor` respectively.

```
let land : bool -> bool -> bool
let lor : bool -> bool -> bool
let not : bool -> bool
let implies : bool -> bool -> bool
```

5.1.3 Integers

`string_of_int` The operator `string_of_int` converts an integer to the corresponding (decimal) string.

```
let string_of_int : int -> string
```

`bgt, blt, bgeq, bleq` These operators are the standard comparisons on integers. Infix operators `gt`, `lt`, `geq`, `leq` resolve to these operators. In this module, we use names like `bgt` here because `gt` is a reserved keyword.

```
let bgt : int -> int -> bool
let blt : int -> int -> bool
let bgeq : int -> int -> bool
let bleq : int -> int -> bool
```

`plus, minus, times, div, mod` These operators are the standard arithmetic functions on integers.

```
let plus : int -> int -> int
let minus : int -> int -> int
let times : int -> int -> int
let div : int -> int -> int
let mod : int -> int -> int
```

5.1.4 Characters

`code` The `code` function converts a `char` to its ASCII code.

```
let code : char -> int
```

`chr` The `chr` function converts an integer in the range `0` to `255` to the corresponding `char`.

```
let chr : (n:int where land (bleq 0 n) (bgeq 255 n)) -> char
```

`string_of_char` The `string_of_char` function converts a character to a string.

```
let string_of_char : char -> string
```

5.1.5 Strings

`length` The `length` function computes the length of a string.

```
let length : string -> int

test length "" = 0
test length "Boomerang" = 9
```

`get_char` The `get_char` function gets a character from a string.

```
let get_char : (s:string ->
               (n:int where land (bleq 0 n) (bgt (length s) n)) ->
               char)

test get_char "Boomerang" 0 = 'B'
test get_char "Boomerang" 1 = 'o'
test get_char "Boomerang" 2 = 'o'
test get_char "Boomerang" 3 = 'm'
test get_char "Boomerang" 8 = 'g'
test get_char "Boomerang" 9 = error
```

`string_concat` The `string_concat` operator is the standard string concatenation function. The overloaded infix `.` operator resolves to `string_concat` when it is applied to strings.

```
let string_concat : string -> string -> string

test string_concat "" "" = ""
test string_concat "Boom" "erang" = "Boomerang"
test string_concat "" "Boomerang" = "Boomerang"
```

5.1.6 Regular Expressions

`str` The `str` function converts a `string` to the singleton `regexp` containing it. This coercion is automatically inserted by the type checker on programs that use subtyping. However, it is occasionally useful to explicitly promote strings to regexps, so we include it here.

```
let str : string -> regexp
```

`EMPTY` The regular expression `empty` denotes the empty set of strings.

```
let EMPTY : regexp = []
```

`EPSILON` The regular expression `epsilon` denotes the singleton set containing the empty string.

```
let EPSILON : regexp = (str "")
```

`string_of_regexp` The `string_of_regexp` function represents a regular expression as a string.

```
let string_of_regexp : regexp -> string
```

`regexp_union` The `regexp_union` operator forms the union of two regular expressions. The overloaded infix symbol `|` desugars into `regexp_union` when used with values of type `regexp`.

```
let regexp_union : regexp -> regexp -> regexp
```

`regexp_concat` The `regexp_concat` operator forms the concatenation of two regular expressions. The overloaded infix symbol `.` desugars into `regexp_concat` when used with values of type `regexp`.

```
let regexp_concat : regexp -> regexp -> regexp
```

`regexp_iter` The `regexp_iter` operator iterates a regular expression. The overloaded symbols `*`, `+`, and `?`, as well as iterations `{n,m}` and `{n,}` all desugar into `regexp_iter` when used with values of type `regexp`. If the second argument is negative, then the iteration is unbounded. For example, `R*` desugars into `regexp_iter R 0 (-1)`.

```
let regexp_iter : regexp -> int -> int -> regexp

let regexp_star (r : regexp) : regexp =
  regexp_iter r 0 (minus 0 1)
```

```
let regexp_plus (r : regexp) : regexp =
  regexp_iter r 1 (minus 0 1)

let regexp_option (r : regexp) : regexp =
  regexp_iter r 0 1
```

`inter` The `inter` operator forms the intersection of two regular expressions. The infix symbol `&` desugars into `inter`.

```
let inter : regexp -> regexp -> regexp
```

`diff` The `diff` operator forms the difference of two regular expressions. The infix symbol `-` desugars into `diff`.

```
let diff : regexp -> regexp -> regexp
```

`representative` The function `representative` computes a (typically shortest) representative of a regular expression.

```
let representative : regexp -> string
```

If the regular expression denotes the empty language, an exception is raised, as the unit test below illustrates.

```
test representative (regexp_iter [A-Z] 1 3) = "A"
test representative [] = error
```

`is_empty` The `is_empty` function tests if a regular expression denotes the empty language.

```
let is_empty : regexp -> bool

test is_empty [] = true
test is_empty [A-Z] = false
test is_empty (diff [A-Z] [^]) = true
```

`equiv,equiv_cex` The `equiv` function tests if two regular expressions denote the same language.

```
let equiv : regexp -> regexp -> bool
let equiv_cex : regexp -> regexp -> bool

test equiv [A-Z] [\065-\090] = true
```

`matches,matches_cex` The `matches` function tests if a string belongs to the language denoted by a regular expression.

```
let matches : regexp -> string -> bool
let matches_cex : regexp -> string -> bool

test matches [A-Z] "A" = true
test matches [A-Z] "0" = false
test matches (diff [^] [A-Z]) "X" = false
test matches (diff [^] [A-Z]) "0" = true
```

`disjoint,disjoint_cex` The `disjoint` function tests whether two regular expressions denote disjoint languages.

```
let disjoint : regexp -> regexp -> bool

let disjoint_cex : regexp -> regexp -> bool

test disjoint [A-Z] [0-9] = true
test disjoint [A-Z] [M] = false
```

`splittable,splittable_cex` The `splittable` function tests whether the concatenation of two regular expressions is ambiguous.

```
let splittable : regexp -> regexp -> bool

let splittable_cex : regexp -> regexp -> bool

test splittable (regexp_iter [A] 0 1) (regexp_iter [A] 0 1) = false
test splittable (regexp_iter [A] 1 1) (regexp_iter [A] 0 1) = true
```

`iterable,iterable_cex` The `iterable` function tests whether the iteration of a regular expression is ambiguous.

```
let iterable : regexp -> bool

let iterable_cex : regexp -> bool

test iterable (regexp_iter [A] 0 1) = false
test iterable (regexp_iter [A] 1 1) = true
```

`count` The `count` function takes as arguments a regular expression `R` and a string `w`. It returns the maximum number of times that `w` can be split into substrings, such that each substring belongs to `R`.

```
let count : (r:regexp ->
             (s:string where matches (regexp_star r) s) ->
            int)

test count [A-Z] "" = 0
test count [A-Z] "ABC" = 3
test count (regexp_option [A-Z]) "ABC" = 3
test count (regexp_option [A-Z]) "123" = error
```

5.1.7 Tags

`species,predicate,key,tag` A tag is a type defined by the Core module and used by the match functions (`aregexp_match` and `lens_match`). The `key_annotation` is used to set the default annotation for the chunk: with `Key` everything without annotations are key, while with `NoKey` they are not.

```
type species = Positional | Diffy of bool | Greedy | Setlike
type predicate = Threshold of (t:int where land (bgeq t 0) (bleq t 100))
type key_annotation = Key | NoKey
type tag = Tag of species * predicate * key_annotation * string

let diffy (name:string) : tag
= Tag (Diffy true, Threshold 0, Key, name)
let positional (name:string) : tag
= Tag (Positional, Threshold 0, NoKey, name)
let greedy (t:int where land (bgeq t 0) (bleq t 100)) (name:string) : tag
= Tag (Greedy, Threshold t, NoKey, name)
let dictionary (name:string) : tag
= Tag (Greedy, Threshold 100, NoKey, name)
let setlike (t:int where land (bgeq t 0) (bleq t 100)) (name:string) : tag
= Tag (Setlike, Threshold t, NoKey, name)
```

5.1.8 Annotated Regular Expressions

`rxlift` The `rxlift` function converts a `regexp` to an equivalent annotated regular expression. This coercion is automatically inserted by the type checker on programs that use subtyping.

```
let rxlift : regexp -> aregexp
```

`rxdrop` The `rxdrop` function drops the annotation of an annotated regular expressions.

```
let rxdrop : aregexp -> regexp
```

```
test equiv (rxdrop (rxlift [a-z])) [a-z] = true
```

`aequiv,aequiv_cex` The `aequiv` function tests if two annotated regular expressions denote the same chunk structured language. It's conservative.

```
let aequiv : aregexp -> aregexp -> bool
let aequiv_cex : aregexp -> aregexp -> bool
```

```
let aregexp_match_compatible_cex : tag -> aregexp -> bool
let aregexp_compatible_cex : aregexp -> aregexp -> bool
```

`aregexp_concat` The `aregexp_concat` operator forms the concatenation of two annotated regular expressions. The overloaded infix symbol **.** desugars into `aregexp_concat` when used with values of type `aregexp`.

```
let aregexp_concat (a1:aregexp)
                   (a2:aregexp where aregexp_compatible_cex a1 a2)
  : aregexp
```

`regexp_union` The `regexp_union` operator forms the union of two regular expressions. The overloaded infix symbol `|` desugars into `regexp_union` when used with values of type `regexp`.

```
let aregexp_union (a1:aregexp)
                  (a2:aregexp where land (disjoint_cex (rxdrop a1) (rxdrop a2))
                                         (aregexp_compatible_cex a1 a2))
  : aregexp
```

`aregexp_iter` The `aregexp_iter` operator iterates an annotated regular expression. The overloaded symbols `*`, `+`, and `?`, as well as iterations `{n,m}` and `{n,}` all desugar into `aregexp_iter` when used with values of type `aregexp`. If the second argument is negative, then the iteration is unbounded. For example, `R*` desugars into `aregexp_iter R 0 (-1)`.

```
let aregexp_iter : aregexp -> int -> int -> aregexp
```

```
let aregexp_star (r : aregexp) : aregexp =
  aregexp_iter r 0 (minus 0 1)
```

```
let aregexp_plus (r : aregexp) : aregexp =
  aregexp_iter r 1 (minus 0 1)
```

```
let aregexp_option (r : aregexp) : aregexp =
  aregexp_iter r 0 1
```

`aregexp_match` The `aregexp_match` function add a chunk annotation with tag defined by the string to the annotated regular expression. The operator `<aregexp>` and `<tag:aregexp>` desugars into `aregexp_match`.

```
let aregexp_match (t:tag) (a:aregexp where aregexp_match_compatible_cex t a)
: aregexp
```

`no_chunks` The `no_chunks` function tests if an annotated regular expressions contains chunk annotations.

```
let no_chunks : aregexp -> bool

test no_chunks (rxlift [a-z]) = true
test no_chunks (aregexp_match (positional "") (rxlift [a-z])) = false
```

5.1.9 Equivalence Relations

`rel` The `rel` datatype splits the equivalence relations on lens (concreate/abstract) domains into two types: identity equivalences, and unknown equivalences.

```
type rel = Identity | Unknown

let rel_is_id (r:rel) : bool =
  equals{rel} r Identity
```

5.1.10 Lens Components

`stype` The `stype` function extracts the dropped concrete type component (i.e., the type of the domain of its *get* function) of a lens. The record-style projection notation `l.stype` and `l.domain_type` both desugar into `stype`.

```
let stype : lens -> regexp
```

`astype` The `astype` function extracts the concrete type component of a lens. The record-style projection notation `l.astype` desugars into `astype`.

```
let astype : lens -> aregexp
```

`vtype` The `vtype` function extracts the dropped abstract type component (i.e., the type of the codomain of its *get* function) of a lens. The record-style projection notation `l.vtype` and `l.codomain_type` both desugar into `vtype`.

```
let vtype : lens -> regexp
```

`avtype` The `avtype` function extracts the abstract type component of a lens. The record-style projection notation `l.avtype` desugars into `avtype`.

```
let avtype : lens -> aregexp
```

`ktype` The `ktype` function extracts the complement type component.

```
let ktype : lens -> skeleton_set
```

`mtype` The `mtype` function extracts the resource type component.

```
let mtype : lens -> resource_set
```

`mtype_compatible_cex` The `mtype_compatible_cex` function returns `true` if the two types can be used for union or concat.

```
let mtype_compatible_cex : resource_set -> resource_set -> bool
```

`mtype_match_compatible_cex` The `mtype_match_compatible_cex` function returns `true` if the tag `t` with the ktype `k` can be used with the mtype `m` for match.

```
let mtype_match_compatible_cex : tag -> skeleton_set -> resource_set -> bool
```

`mtype_domain_equal` The `mtype_domain_equal` function returns `true` if the two types have the same domain. It's used for compose.

```
let mtype_domain_equal : resource_set -> resource_set -> bool
```

`vrep,srep`

```
let vrep : lens -> string -> string
let srep : lens -> string -> string
```

`sequiv` The `sequiv` function extracts the equivalence relation on the concrete domain (`astype`) of a lens.

```
let sequiv : lens -> rel
```

`vequiv` The `vequiv` function extracts the equivalence relation on the abstract domain (`avtype`) of a lens.

```
let vequiv : lens -> rel
```

`bij` The `bij` function tests whether a lens is bijective. The record-style projection notation `l.bij` desugars into `bij`.

```
let bij : lens -> bool
```

`is_basic` The `is_basic` function tests whether a lens is a basic lens (i.e. does not contain any chunk).

```
let is_basic (l:lens) : bool =
  no_chunks (avtype l)
```

`in_lens_type` The `in_lens_type` function tests whether a lens is in a given `stype` and `vtype`. The `lens in S <-> V` notation desugars into `in_lens_type`.

```
let in_lens_type (l:lens) (s:regexp) (v:regexp) : bool =
  (land (equiv_cex (stype l) s) (equiv_cex (vtype l) v))
```

`in_bij_lens_type` The `in_lens_type` function tests whether a lens is bijective in a given `stype` and `vtype`. The `lens in S <=> V` notation desugars into `in_bij_lens_type`.

```
let in_bij_lens_type (l:lens) (S:regexp) (V:regexp) : bool =
  (land (land (equiv_cex (stype l) S) (equiv_cex (vtype l) V)) (bij l))
```

`get` The `get` function extracts the *get* component of a lens. The record-style projection notation `l.get` desugars into `get`.

```
let get : (l:lens ->
          (s:string where matches_cex (stype l) s) ->
          string)
```

`put` The `put` function extracts the *put* component of a lens. The record-style projection notation `l.put v into s` desugars into `put`.

```
let put : (l:lens ->
          (v:string where matches_cex (vtype l) v) ->
          (s:string where matches_cex (stype l) s) ->
          string)
```

`create` The `create` function extracts the *create* component of a lens. The record-style projection notation `l.create` desugars into `create`.

```
let create : (l:lens ->
             (v:string where matches_cex (vtype l) v) ->
             string)
```

5.1.11 Lenses

`copy` The `copy` lens takes a regular expression `R` as an argument and copies strings belonging to `R` in both directions.

```
let copy (R:regexp) : (l:lens where in_bij_lens_type l R R)

test get (copy [A-Z]) "A" = "A"
test put (copy [A-Z]) "B" "A" = "B"
test create (copy [A-Z]) "Z" = "Z"
test get (copy [A-Z]) "1" = error
test stype (copy [A-Z]) = [A-Z]
test vtype (copy [A-Z]) = stype (copy [A-Z])
```

`clobber` The `clobber` lens takes as arguments a regular expression `R`, a string `u`, and a function from strings to strings `f`. Its *get* function is the constant function that returns `u`, its *put* function restores its concrete argument, and its *create* function returns the string `f u`.

```
let clobber
    (R:regexp) (u:string) (f:string -> (s:string where matches R s))
  : (l:lens where in_lens_type l R (str u))

test get (clobber [A-Z] "" (fun (s:string) -> "B")) "A" = ""
test put (clobber [A-Z] "" (fun (s:string) -> "B")) "" "A" = "A"
test create (clobber [A-Z] "" (fun (s:string) -> "B")) "" = "B"
```

`const` The `const` lens behaves like `clobber` but has a *create* function that always returns a default string `v`.

```
let const (R:regexp) (u:string) (v:string where matches R v)
  : (l:lens where in_lens_type l R (str u))

test get (const [A-Z] "x" "B") "A" = "x"
test put (const [A-Z] "x" "B") "x" "A" = "A"
test create (const [A-Z] "x" "B") "x" = "B"
```

`set` The `set` derived lens is like `const` but uses an arbitrary representative of `R` as the default string. The infix operator `<->` desugars to `set`.

```
let set (R:regexp) (s:string) : (l:lens where in_lens_type l R (str s))
  = const R s (representative R)
```

`rewrite` The `rewrite` derived lens is like `set` but only rewrites strings, and so is bijective. The infix operator <=> desugars to `rewrite`.

```
let rewrite (s1:string) (s2:string)
  : (l:lens where in_bij_lens_type l (str s1) (str s2))
  = const (str s1) s2 s1
```

`lens_union` The `lens_union` operator forms the union of two lenses. The concrete types of the two lenses must be disjoint. The overloaded infix operator `||` desugars into `lens_union` when applied to lens values.

```
let lens_union
    (l1:lens where land (rel_is_id (vequiv l1))
                        (is_basic l1))
    (l2:lens where land (rel_is_id (vequiv l2))
                  (land (is_basic l2)

                        (disjoint_cex (stype l1) (stype l2))))
  : (l:lens where in_lens_type l
         (regexp_union (stype l1) (stype l2))
         (regexp_union (vtype l1) (vtype l2)))

test get (lens_union (copy [A-Z]) (copy [0-9])) "A" = "A"
test get (lens_union (copy [A-Z]) (copy [0-9])) "0" = "0"
test create (lens_union (copy [A-Z]) (copy [0-9])) "A" = "A"
test lens_union (copy [A-Z]) (copy [^]) = error
```

`lens_disjoint_union` The `lens_disjoint_union` operator also forms the union of two lenses. However, it requires that the concrete and abstract types of the two lenses be disjoint. The overloaded infix operator `|` desugars into `lens_disjoint_union` when applied to lens values.

```
let lens_disjoint_union_contract (l1:lens) (l2:lens) : bool
= land (mtype_compatible_cex (mtype l1) (mtype l2))
       (land (disjoint_cex (stype l1) (stype l2))
             (disjoint_cex (vtype l1) (vtype l2)))

let lens_disjoint_union
    (l1:lens)
    (l2:lens where lens_disjoint_union_contract l1 l2)
  : (l:lens where in_lens_type l
         (regexp_union (stype l1) (stype l2))
         (regexp_union (vtype l1) (vtype l2)))

test get (lens_disjoint_union (copy [A-Z]) (copy [0-9])) "A" = "A"
test get (lens_disjoint_union (copy [A-Z]) (copy [0-9])) "0" = "0"
test lens_disjoint_union (copy [A-Z]) (const [0-9] "A" "0") = error
```

`lens_concat` The `lens_concat` operator forms the concatenation of two lenses. The concrete and abstract types of the two lenses must each be unambiguously concatenable. The overloaded infix operator . desugars into `lens_concat` when applied to lens values.

```
let lens_concat_contract (l1:lens) (l2:lens) : bool
= land (mtype_compatible_cex (mtype l1) (mtype l2))
       (land (splittable_cex (stype l1) (stype l2))
             (splittable_cex (vtype l1) (vtype l2)))

let lens_concat
    (l1:lens)
    (l2:lens where lens_concat_contract l1 l2)
  : (l:lens where in_lens_type l (regexp_concat (stype l1) (stype l2))
                                 (regexp_concat (vtype l1) (vtype l2)))

test get (lens_concat (copy [A-Z]) (copy [0-9])) "A1" = "A1"
test put (lens_concat (copy [A-Z]) (copy [0-9])) "B2" "A1" = "B2"
test create (lens_concat (copy [A-Z]) (copy [0-9])) "B2" = "B2"
```

`compose` The `compose` operator puts two lenses in sequence. The abstract type of the lens on the left and the concrete type of the lens on the right must be identical.

```
let compose
    (l1:lens where rel_is_id (vequiv l1))
    (l2:lens where land (aequiv (avtype l1) (astype l2))
                        (rel_is_id (sequiv l2)))
  : (l:lens where in_lens_type l (stype l1) (vtype l2))

test get (compose (const [A-Z] "Z" "A")
                  (const [Z] "X" "Z")) "A" = "X"
```

`lens_swap` The `lens_swap` operator also concatenates lenses. However, it swaps the order of the strings it creates on the abstract side. As with `lens_concat`, the concrete and abstract types of the two lenses must each be unambiguously concatenable. The overloaded infix operator ~ desugars into `lens_swap` when applied to lens values.

```
let lens_swap_contract (l1:lens) (l2:lens) : bool
= land (mtype_compatible_cex (mtype l1) (mtype l2))
       (land (splittable_cex (stype l1) (stype l2))
             (splittable_cex (vtype l2) (vtype l1)))

let lens_swap
    (l1:lens)
    (l2:lens where lens_swap_contract l1 l2)
  : (l:lens where in_lens_type l (regexp_concat (stype l1) (stype l2))
                                 (regexp_concat (vtype l2) (vtype l1)))
```

```
test get (lens_swap (copy [A-Z]) (copy [0-9])) "A1" = "1A"
test put (lens_swap (copy [A-Z]) (copy [0-9])) "2B" "A1" = "B2"
test create (lens_swap (copy [A-Z]) (copy [0-9])) "2B" = "B2"
```

`lens_star` The `lens_star` operator iterates a lens zero or more times. The iterations of the concrete and abstract types of the lens must both be unambiguous. The overloaded operator $\star$ desugars into `lens_star` when applied to a lens. Recall that `regexp_iter R 0 -1` is how `R*` desugars.

```
let lens_star_contract (l:lens) : bool
= land (iterable_cex (stype l)) (iterable_cex (vtype l))

let lens_star
    (l:lens where lens_star_contract l)
  : (l':lens where in_lens_type l'
               (regexp_star (stype l))
               (regexp_star (vtype l)))

test get (lens_star (copy [A-Z])) "" = ""
test get (lens_star (copy [A-Z])) "ABCD" = "ABCD"
test put (lens_star (copy [A-Z])) "AB" "ABCD" = "AB"
test create (lens_star (copy [A-Z])) "A" = "A"
```

`lens_plus` The `lens_plus` operator iterates a lens one or more times. The iterations of the concrete and abstract types of the lens must both be unambiguous (when non-empty). The overloaded operator + resolves to `lens_plus` when applied to a lens.

```
let lens_plus
    (l:lens where land (iterable_cex (stype l))
                       (iterable_cex (vtype l)))
  : (l':lens where in_lens_type l'
               (regexp_plus (stype l))
               (regexp_plus (vtype l)))

test get (lens_plus (copy [A-Z])) "A" = "A"
test get (lens_plus (copy [A-Z])) "ABCD" = "ABCD"
test put (lens_plus (copy [A-Z])) "AB" "ABCD" = "AB"
test create (lens_plus (copy [A-Z])) "A" = "A"
```

`lens_option` The `lens_option` operator runs a lens once or not at all. The concrete and abstract types of the lens must both be disjoint from the empty lens. The overloaded operator `?` resolves to `lens_option` when applied to a lens.

```
let lens_option
    (l:lens where land (disjoint_cex (stype l) EPSILON)
                       (disjoint_cex (vtype l) EPSILON))
  : (l':lens where in_lens_type l'
               (regexp_union (stype l) EPSILON)
               (regexp_union (vtype l) EPSILON))

test get (lens_option (copy [A-Z])) "" = ""
test get (lens_option (copy [A-Z])) "A" = "A"
test put (lens_option (copy [A-Z])) "B" "" = "B"
test create (lens_option (copy [A-Z])) "A" = "A"
```

`lens_iter` The `lens_iter` operator iterates a lens a finite number of times. The concatenations of the concrete and abstract types of the lens must both be unambiguous. The overloaded operator `{m,n}` resolves into instances of `lens_iter` when applied to a lens argument.

```
let lens_iter
   (l:lens where land (splittable_cex (stype l) (stype l))
                      (splittable_cex (vtype l) (vtype l)))
   (min:int where bgeq min 1)
   (max:int where
      land (bgeq max min)
           (implies (bgt max min)
             (land (disjoint_cex (stype l) (regexp_iter (stype l) 2 2))
             (lor (land (rel_is_id (vequiv l)) (is_basic l))
                  (disjoint_cex (vtype l) (regexp_iter (vtype l) 2 2))))))
 : (l':lens where in_lens_type l'
             (regexp_iter (stype l) min max)
             (regexp_iter (vtype l) min max))

test get (lens_iter (copy [A-Z]) 1 4) "ABCD" = "ABCD"
test put (lens_iter (copy [A-Z]) 1 4) "AB" "ABCD" = "AB"
test create (lens_iter (copy [A-Z]) 1 4) "A" = "A"
```

`invert` The `invert` operator swaps the *get* and *create* components of a lens, which must be bijective.

```
let invert
    (l:lens where land (bij l) (is_basic l))
  : (l':lens where in_bij_lens_type l' (vtype l) (stype l))

test get (invert (const [A] "B" "A")) "B" = "A"
test invert (const [A-Z] "B" "A") = error
```

`default` The `default` operator takes a lens `l` and a string `d` as arguments. It overrides `l`'s *create* function to use *put* with `d`.

```
let default (l:lens where is_basic l)
            (d:string where matches (stype l) d)
          : (l':lens where in_lens_type l' (stype l) (vtype l))

test create (default (const [A-Z] "X" "A") "B") "X" = "B"
```

`partition` The `partition` operator takes two regular expressions `R` and `S` as arguments and produces a lens whose *get* function transforms a string belonging to the iteration of the union of `R` and `S` by sorting the substrings into substrings that belong to `R` and `S`. The regular expressions `R` and `S` must be disjoint and the iteration of their union must be unambiguous.

```
let partition
    (R:regexp)
    (S:regexp where land (disjoint_cex R S)
                         (iterable_cex (regexp_union R S)))
  : (l:lens where in_lens_type l
                  (regexp_star (regexp_union R S))
                  (regexp_concat
                    (regexp_star R)
                    (regexp_star S)))

test get (partition [A-Z] [0-9]) "A1B2C3" = "ABC123"
test put (partition [A-Z] [0-9]) "ABC123456" "A1B2C3" = "A1B2C3456"
```

`merge` The `merge` operator takes a regular expression `R` and produces a lens whose *get* function transforms a string belonging to the concatenation of `R` with itself by discarding the second substring belonging to `R`. The regular expression `R` must be unambiguously concatenable with itself.

```
let merge
    (R:regexp where splittable_cex R R)
  : (l:lens where in_lens_type l (regexp_concat R R) R)

test get (merge [A-Z]) "AA" = "A"
test get (merge [A-Z]) "AB" = "A"
test put (merge [A-Z]) "C" "AA" = "CC"
test put (merge [A-Z]) "C" "AB" = "CB"
```

5.1.12 Resourceful Lenses

`key,nokey,force_key,force_nokey` These functions defines the key annotation of the characters under it. The `force_key` and `force_nokey` overrides previous definitions while `key` and `nokey` only set the annotation for characters that does not have yet an annotation.

```
let key (l:lens)
     : (l':lens where in_lens_type l' (stype l) (vtype l))

let nokey (l:lens)
       : (l':lens where in_lens_type l' (stype l) (vtype l))

let force_key (l:lens)
            : (l':lens where in_lens_type l' (stype l) (vtype l))

let force_nokey (l:lens)
              : (l':lens where in_lens_type l' (stype l) (vtype l))
```

`lens_match` The `lens_match` operator takes as arguments a string `t` and a lens `l` and creates a "chunk" with tag `t`. The type checker requires that there is not a tag inside `l` with the same identifier as the tag `t`. The operator `<tag:aregexp>` desugars into `lens_match` and the operator `<aregexp>` desugars to `lens_match` with tag `greedy 0 ""`.

```
let lens_match
    (t:tag)
    (l:lens where mtype_match_compatible_cex t (ktype l) (mtype l))
  : (l':lens where in_lens_type l' (stype l) (vtype l))
```

`align` The `align` operator converts a resourcefull lens `l` into a basic lens, making an alignment phase internal to it.

```
let align
    (l:lens)
  : (l':lens where land (in_lens_type l' (stype l) (vtype l))
                        (is_basic l'))
```

`fiat` The `fiat` operator takes a lens `l` as an argument. It behaves like `l`, but overrides its *put* component with a function that returns the original source exactly whenever the update to the view is a no-op.

```
let fiat
    (l:lens where is_basic l)
  : (l':lens where in_lens_type l' (stype l) (vtype l))
```

5.1.13 Canonizer Components

`uncanonized_type` The `uncanonized_type` function extracts the "representative" type component (i.e., the type of the domain of its *canonize* function) of a canonizer.

```
let uncanonized_type : canonizer -> regexp

let uncanonized_atype : canonizer -> aregexp
```

`canonized_type` The `canonized_type` function extracts the "quotiented" type component (i.e., the type of the codomain of its *canonize* function) of a canonizer.

```
let canonized_type : canonizer -> regexp

let canonized_atype : canonizer -> aregexp
```

`in_canonizer_type` The `in_canonizer_type` function tests whether a canonizer has the given uncanonized and canonized types.

```
let in_canonizer_type (cn:canonizer) (U:regexp) (C:regexp)
  = (land (equiv_cex (uncanonized_type cn) U)
          (equiv_cex (canonized_type cn) C))
```

`canonizer_is_basic` The `canonizer_is_basic` function tests whether a canonizer is a basic canonizer (i.e. does not contain any chunk).

```
let canonizer_is_basic (cn:canonizer) : bool =
  no_chunks (canonized_atype cn)
```

5.1.14 Canonizers

`cnrel` The `cnrel` function extracts the equivalence relation on a canonizer's (canonized) type.

```
let cnrel : canonizer -> rel
```

`canonize` The `canonize` function extracts the *canonize* component of a canonizer. The record-style projection notation `q.canonize` desugars into `canonize`.

```
let canonize
    (cn:canonizer)
    (c:string where matches (uncanonized_type cn) c)
  : string
```

`choose` The `choose` function extracts the *choose* component of a canonizer. The record-style projection notation `q.choose` desugars into `choose`.

```
let choose
    (cn:canonizer)
    (b:string where matches (canonized_type cn) b)
  : string
```

`canonizer_of_lens` The `canonizer_of_lens` operator builds a canonizer out of a lens with the lens's *get* function as the *canonize* component and *create* as *choose*.

```
let canonizer_of_lens (l:lens)
  : (cn:canonizer where in_canonizer_type cn (stype l) (vtype l))
```

`canonizer_concat` The `canonizer_concat` operator concatenates canonizers. Only the concatenation of types on the left side needs to be unambiguous.

```
let canonizer_concat
  (cn1:canonizer)
  (cn2:canonizer where
    land (splittable_cex (uncanonized_type cn1) (uncanonized_type cn2))
         (implies
            (not (land (rel_is_id (cnrel cn1)) (rel_is_id (cnrel cn2))))
            (splittable_cex (canonized_type cn1) (canonized_type cn2))))
  : (cn:canonizer where in_canonizer_type cn
          (regexp_concat (uncanonized_type cn1) (uncanonized_type cn2))
          (regexp_concat (canonized_type cn1) (canonized_type cn2)))
```

`canonizer_union` The `canonizer_union` operator forms the union of two canonizers. The types on the left need to be disjoint.

```
let canonizer_union
    (cn1:canonizer)
    (cn2:canonizer where land (disjoint_cex (uncanonized_type cn1)
                                            (uncanonized_type cn2))
                              (lor (land (canonizer_is_basic cn1)
                                         (canonizer_is_basic cn2))
                                   (disjoint_cex (canonized_type cn1)
                                                 (canonized_type cn2))))
  : (cn:canonizer where in_canonizer_type cn
        (regexp_union (uncanonized_type cn1) (uncanonized_type cn2))
        (regexp_union (canonized_type cn1) (canonized_type cn2)))
```

`canonizer_iter` The `canonizer_iter` operator iterates a canonizer. The iteration of the type on the left needs to be unambiguous. The overloaded operators `*`, `+`, `?`, `{m,n}` and `{n,}` all desugar into instances of `canonizer_iter` when applied to a canonizer. If the second argument is negative, then the iteration is unbounded. For example, `q*` desugars into `canonizer_iter q 0 (-1)`.

```
let canonizer_iter
  (cn:canonizer where
       land (iterable_cex (uncanonized_type cn))
            (implies (not (rel_is_id (cnrel cn)))
                     (iterable_cex (canonized_type cn))))
  (min:int where bgeq min 0) (max:int)
: (cn':canonizer where in_canonizer_type cn'
        (regexp_iter (uncanonized_type cn) min max)
        (regexp_iter (canonized_type cn) min max))
```

`columnize` The `columnize` primitive canonizer wraps long lines of text. It takes as arguments an integer `n`, a regular expression `R`, a character `s` and a string `nl`. It produces a canonizer whose *canonize* component takes strings belonging to the iteration of `R`, extended so that `s` and `nl` may appear anywhere that `s` may, and replaces `nl` with `s` globally. Its *choose* component wraps a string belonging to the iteration of `R` by replacing `s` with `nl` to obtain a string in which (if possible) the length of every line is less than or equal to `n`.

```
let columnize
  (k:int)
  (R:regexp)
  (sp:char)
  (nl:string where
        disjoint_cex R (regexp_concat (regexp_star [^])
                                      (regexp_concat (str nl)
                                      (regexp_star [^]))))
: canonizer
```

The following unit test illustrates the *choose* component of `columnize`.

```
test choose (columnize 5 (regexp_star [a-z ]) ' ' "\n")
      "a b c d e f g" =
<<
  a b c
  d e f
  g
>>
```

5.1.15 Quotient Lenses

The next few primitives construct lenses that work up to programmer-specified equivalence relations. We call these structures quotient lenses. For details, see Foster et al. [2008].

`left_quot` The `left_quot` operator quotients a lens `l` by a canonizer `q` on the left by passing concrete strings through `q`.

```
let left_quot
  (cn:canonizer)
  (l:lens where land (aequiv (canonized_atype cn) (astype l))
                     (rel_is_id (cnrel cn)))
: (l':lens where in_lens_type l' (uncanonized_type cn) (vtype l))

test get
  (left_quot (columnize 5 (regexp_star [a-z ]) ' ' "\n")
             (copy (regexp_star [a-z ])))
<<
  a b c
  d e f
  g
>>
= "a b c d e f g"

test create
  (left_quot (columnize 5 (regexp_star [a-z ]) ' ' "\n")
             (copy (regexp_star [a-z ])))
"a b c d e"
=
<<
  a b c
  d e
>>
```

`right_quot` The `right_quot` operator quotients a lens `l` by a canonizer `q` on the right by passing abstract strings through `q`.

```
let right_quot
  (l:lens)
  (cn:canonizer where land (aequiv (canonized_atype cn) (avtype l))
                           (rel_is_id (cnrel cn)))
: (l':lens where in_lens_type l' (stype l) (uncanonized_type cn))
```

`dup1` The `dup1` operator takes as arguments a lens `l`, a function `f`, and a regular expression `R`, which should denote the codomain of `f`. Its *get* function supplies one copy of the concrete string to `l`'s *get* function and one to `f`, and concatenates the results. The *put* and *create* functions simply discard the part of the string computed by `f` and use the corresponding from `l` on the rest of the string. The concatenation of `l`'s abstract type and the codomain of `f` must be unambiguous.

```
let dup1
    (l:lens where is_basic l)
    (R:regexp)
    (f:string -> (x:string where matches R x))
 :  (l':lens where in_lens_type l'
          (stype l) (regexp_concat (vtype l) R))

test get (dup1 (copy [A-Z]) [A-Z] (get (copy [A-Z]))) "A" = "AA"
test put (dup1 (copy [A-Z]) [A-Z] (get (copy [A-Z]))) "BC" "A" = "B"
```

`dup2` The `dup2` operator is like `dup1` but uses `f` to build the first part of the output.

```
let dup2
    (R:regexp)
    (f:string -> (x:string where matches R x))
    (l:lens where is_basic l)
  : (l':lens where in_lens_type l'
        (stype l) (regexp_concat R (vtype l)))

test get (dup2 [A-Z] (get (copy [A-Z])) (copy [A-Z])) "A" = "AA"
test put (dup2 [A-Z] (get (copy [A-Z])) (copy [A-Z])) "BC" "A" = "C"
```

5.2 The Standard Prelude

The second module, `Prelude`, defines some common derived forms. Like `Core`, its values are available by default in every Boomerang program.

5.2.1 Regular Expressions

`ANYCHAR, ANY, ANYP` The regular expression `ANYCHAR` denotes the set of ASCII characters, `ANY` denotes the set of all ASCII strings, and `ANYP` denotes the set of all ASCII strings except for the empty string. By convention, we append a "P" to the name of a regular expression to denote its "positive" variant (i.e., not containing the empty string).

```
let ANYCHAR : regexp = [^]
let ANY : regexp = ANYCHAR*
let ANYP : regexp = ANYCHAR+
```

`containing,containingP,not_containing,not_containingP` The function `containing` takes a regular expression `R` as an argument and produces a regular expression describing the set of all strings that contain a substring described by `R`. The `-P` variants require non-empty strings.

```
let containing  (R:regexp) : regexp = ANY . R . ANY
let containingP (R:regexp) : regexp = (ANY . R . ANY) - []
```

The function `not_containing` takes a regular expression `R` and produces a regular expression describing the set of all strings not containing `R`.

```
let not_containing  (R:regexp) : regexp = ANY - (containing R)
let not_containingP (R:regexp) : regexp = ANYP - (containing R)
```

`SCHAR, S, SP` The regular expressions `SCHAR`, `S`, and `SP` denote sets of space characters.

```
let SCHAR : regexp = [ ]
let S : regexp = SCHAR*
let SP : regexp = SCHAR+
```

`WSCHAR, WS, WSP` The regular expressions `WSCHAR`, `WS`, and `WSP` denote sets of whitespace characters.

```
let WSCHAR : regexp = [ \t\r\n]
let WS : regexp = WSCHAR*
let WSP : regexp = WSCHAR+
```

`NWSCHAR, NWS, NWSP` The regular expressions `WSCHAR`, `WS`, and `WSP` denote sets of non-whitespace characters.

```
let NWSCHAR : regexp = [^ \t\r\n]
let NWS : regexp = NWSCHAR*
let NWSP : regexp = NWSCHAR+
```

`newline, NLn` The string `newline` contains the newline character. The strings given by `NL`n each denote a newline followed by n spaces. These are used for indentation, for example, in the `Xml` module.

```
let newline : string = "\n"
let NL0 : string= newline
let NL1 : string= NL0 . " "
let NL2 : string= NL1 . " "
let NL3 : string= NL2 . " "
```

```
let NL4 : string= NL3 . " "
let NL5 : string= NL4 . " "
let NL6 : string= NL5 . " "
let NL7 : string= NL6 . " "
let NL8 : string= NL7 . " "
let NL9 : string= NL8 . " "
let NL10 : string = NL9 . " "
```

`DIGIT, NUMBER, FNUMBER` The regular expressions `DIGIT`, `NUMBER`, and `FNUMBER` represent strings of decimal digits, integers, and floating point numbers respectively.

```
let DIGIT : regexp = [0-9]
let NUMBER : regexp = [0] | [1-9] . DIGIT*
let FNUMBER : regexp = NUMBER . ([.] . DIGIT+)?
```

`UALPHACHAR, UALPHANUMCHAR` The regular expression `UALPHACHAR` and `UALPHANUMCHAR` denote the set of upper case alphabetic and alphanumeric characters respectively.

```
let UALPHACHAR : regexp = [A-Z]
let UALPHANUMCHAR : regexp = [A-Z0-9]
```

`is_cset` The predicate `is_cset` on regular expressions determines whether a regular expression identifies a set of characters.

```
let is_cset (R:regexp) : bool = equiv_cex R (R & ANYCHAR)
```

`subset` The binary predicate `subset` decides whether the first regular expression is a subset of the second.

```
let subset (R1:regexp) (R2:regexp) : bool
  = equiv_cex (R1 & R2) R1
```

5.2.2 Lenses

`lens_equiv` The binary operator on lenses, `lens_equiv`, tests whether the `astype`s and `avtype`s of two lenses are equivalent regular expressions (according to `equiv`).

```
let lens_equiv (l1:lens) (l2:lens) : bool =
 (equiv_cex (stype l1) (stype l2)) &&
 (equiv_cex (vtype l1) (vtype l2))
```

`ins` The lens `ins` maps the empty concrete string to a fixed abstract string. It is defined straightforwardly using `<->`.

```
let ins (s:string) : (lens in EPSILON <=> s) = "" <-> s
test get (ins "ABC") "" = "ABC"
test put (ins "ABC") "ABC" "" = ""
```

`del` The lens `del` deletes a regular expression. It is also defined using `<->`.

```
let del (R:regexp) : (lens in R <-> EPSILON) = R <-> ""

test get (del ANY) "Boomerang" = ""
test put (del ANY) "" "Boomerang" = "Boomerang"
test create (del ANY) "" = ""
```

`filter` The `filter` operator takes two regular expressions `R` and `S` as arguments and produces a lens whose *get* function transforms a string belonging to the iteration of the union of `R` and `S` by discarding all of the substrings belonging to `R`. The regular expressions `R` and `S` must be disjoint and the iteration of their union must be unambiguous.

```
let filter
    (R:regexp)
    (S:regexp where (disjoint_cex R S) && (iterable_cex (R | S)))
  : (lens in (R | S)* <-> S* )
  = partition R S; ( del R* . copy S* )

test get (filter [A-Z] [0-9]) "A1B2C3" = "123"
test put (filter [A-Z] [0-9]) "123456" "A1B2C3" = "A1B2C3456"
```

`merge_with_sep` The lens `merge_with_sep` behaves like `merge`, but allows a separator between the copied string.

```
let merge_with_sep (R:regexp) (s:string) : (lens in (R . s . R) <-> R) =
  copy (R . s . R) . ins s;
  merge (R . s);
  copy R . del s

test (merge_with_sep [A-Z] ",").get "A,B" = "A"
test (merge_with_sep [A-Z] ",").put "C" into "A,B" = "C,B"
test (merge_with_sep [A-Z] ",").create "B" = "B,B"
```

5.2.3 Lens Predicates

`lens_iterable` This predicate is true for lenses with iterable stype and vtype.

```
let lens_iterable (l:lens) : bool =
  iterable_cex (stype l) && iterable_cex (vtype l)
```

`lens_splittable` This predicate is true for a pair of lenses if the stypes and vtypes are splittable.

```
let lens_splittable (l1:lens) (l2:lens) : bool =
  splittable_cex (stype l1) (stype l2) &&
  splittable_cex (vtype l1) (vtype l2)
```

`lens_unionable,lens_disjoint` This predicate is true for a pair of lenses if the astypes are disjoint and the equivalence relation abstract domain of the second lens is the identity relation.

```
let lens_unionable (l1:lens) (l2:lens) : bool =
  disjoint_cex (stype l1) (stype l2) &&
  rel_is_id (vequiv l2) && is_basic l1 && is_basic l2

let lens_disjoint (l1:lens) (l2:lens) : bool =
  disjoint_cex (stype l1) (stype l2) &&
  disjoint_cex (vtype l1) (vtype l2)
```

5.2.4 Quotient Lenses

`qconst` The lens `qconst` is like `const`, but accepts an entire regular expression on the abstract side. It is defined using quotienting on the right, the lens `const`, and a canonizer built from `const`.

```
let qconst (C:regexp) (A:regexp) (a:string where matches A a) (c:string where ma
  : (lens in C <-> A)
  =
  right_quot
    (const C a c)
    (canonizer_of_lens (const A a a))

test get (qconst [A-Z] [a-z] "a" "A") "A" = "a"
test put (qconst [A-Z] [a-z] "a" "A") "b" "B" = "B"
```

`qset` The lens `qconst` is like `set` (i.e., `<->`), but takes an entire regular expression on the abstract side. It is defined using `qconst`.

```
let qset (C:regexp) (A:regexp) : (lens in C <-> A) =
 qconst C A (representative A) (representative C)

test get (qset [A-Z] [a-z]) "A" = "a"
test get (qset [A-Z] [a-z]) "Z" = "a"
test put (qset [A-Z] [a-z]) "z" "A" = "A"
test put (qset [A-Z] [a-z]) "z" "Z" = "Z"
```

`qins,qins_representative` The lens `qins` is like `ins` but accepts a regular expression in the *put* direction. It is defined using right quotienting and `ins`. The lens `qins_representative` is +similar, but uses an arbitrary representative of `E` in the *get* direction.

```
let qins (E:regexp) (e:string in E) : (lens in EPSILON <-> E) =
  right_quot
    (ins e)
    (canonizer_of_lens (const E e e))

test (get (qins [A-Z]+ "A") "") = "A"
test (create (qins [A-Z]+ "A") "ABC") = ""

let qins_representative (E:regexp) : (lens in EPSILON <-> E) =
  qins E (representative E)

test (get (qins_representative [A-Z]+) "") = "A"
test (create (qins_representative [A-Z]+) "ABC") = ""
```

`qdel` The lens `qdel` is like `del` but produces a canonical representative in the backwards direction. It is defined using left quotienting.

```
let qdel (E:regexp) (e:string) : (lens in E <-> EPSILON) =
  left_quot
    (canonizer_of_lens (default (del E) e))
    (copy EPSILON)

test (get (qdel [A-Z]+ "ZZZ") "ABC") = ""
test (put (qdel [A-Z]+ "ZZZ") "" "ABC") = "ZZZ"
test (put (qdel [A-Z]+ "ZZZ") "1" "ABC") = error
```

5.2.5 Standard Datatypes

`'a option, ('a,'b) maybe` The polymorphic datatypes `option` and `maybe` represents optional and alternative values respectively.

```
type 'a option =
    None | Some of 'a

type ('a,'b) maybe =
    Left of 'a | Right of 'b
```

5.2.6 Pairs

`fst,snd` The polymorphic functions fst and snd are the standard projections on pairs.

```
let fst ('a) ('b) (p:'a * 'b) : 'a =
  let x,_ = p in x

let snd ('a) ('b) (p:'a * 'b) : 'b =
  let _,y = p in y
```

5.2.7 Lists of Lenses and Regular Expressions

`astypes,avtypes` Calculates the astypes and avtypes of lists of lenses.

```
let astypes (ls:lens List.t) = List.map{lens}{aregexp} astype ls
let avtypes (ls:lens List.t) = List.map{lens}{aregexp} avtype ls
let stypes (ls:lens List.t) = List.map{lens}{regexp} stype ls
let vtypes (ls:lens List.t) = List.map{lens}{regexp} vtype ls
```

`concatable` The list of regexps Rs = #{regexp}[R1;R2;...;Rn] are concatenable with regexp separator S if the following are splittable:

- R1 and S
- R1.S and R2
- R1.S.R2.S and R3
- ...
- R1.S...SRn-1 and Rn

```
let concatable (rl : regexp List.t) : bool =

test concatable #{regexp}["abc";"def"] = true
test concatable #{regexp}["abc";"def"] = true
test concatable #{regexp}["a" | "aa";"a"?] = false
```

`concat_regexps,concat_lenses` Concatenates a list of regular expressions or lenses, respectively.

```
let concat_regexps (Rs:regexp List.t) : regexp =
 List.fold_left{regexp}{regexp}
   (fun (acc:regexp) (R:regexp) -> acc . R)
   EPSILON Rs
```

```
test concat_regexps #{regexp}["abc";"def"] = "abcdef"
test concat_regexps #{regexp}["a"{5};"a"*] = "a"{5,}

let concat_lenses (ls:lens List.t where (concatable (stypes ls))
                                     && (concatable (vtypes ls)))
  : (lens in concat_regexps (stypes ls)
         <-> concat_regexps (vtypes ls))
  =  List.fold_left{lens}{lens}
      (fun (l_acc:lens) (l:lens) -> l_acc . l)
      (copy EPSILON) ls

test get (concat_lenses #{lens}[copy "a";copy "b";copy "c"]) "abc" = "abc"
```

`disjoint_from_regexps,disjoint_regexps` The function `disjoint_from_regexps` determines whether a given regexp is disjoint from a list of regular expressions. The function `disjoint_regexps` determines whether a list of regular expressions are pairwise disjoint.

```
let disjoint_from_regexps (R:regexp) (Rs:regexp List.t) =
  List.fold_left{regexp}{bool}
    (fun (ok:bool) (R':regexp) ->
ok && disjoint_cex R R')
    true Rs

let disjoint_regexps (Rs:regexp List.t) : bool =
  let (ok,_) = List.fold_left{regexp}{bool * regexp List.t}
    (fun (acc:bool * regexp List.t) (R:regexp) ->
       let (ok,Rs) = acc in
       (ok && disjoint_from_regexps R Rs,List.Cons{regexp}(R,Rs)))
    (true,#{regexp}[]) Rs in
  ok

test disjoint_regexps #{regexp}["a";"b";"c"] = true
```

`union_regexps,union_lenses,disj_union_lenses` Takes the union of a list of regular expressions or lenses, respectively. By default, the non-disjoint lens union `||` is used; use `disj_union_lenses` for disjoint union `|`.

```
let union_regexps (Rs:regexp List.t) : regexp =
 List.fold_left{regexp}{regexp}
   (fun (acc:regexp) (R:regexp) -> acc | R)
   EMPTY Rs

test union_regexps #{regexp}["abc";"def"] = ("abc" | "def")
test union_regexps #{regexp}["a"{5};"a"*] = ("a"{5} | "a"* )
```

```
let union_lenses (ls:lens List.t where disjoint_regexps (stypes ls))
  : (lens in union_regexps (stypes ls)
        <-> union_regexps (vtypes ls))
  =
  List.fold_left{lens}{lens}
  (fun (l_acc:lens) (l:lens) -> l_acc || l)
  (copy EMPTY) ls

test create (union_lenses #{lens}["z" <-> "a"; (copy [a-c])]) "a" = "z"
test get (union_lenses #{lens}[copy "a";copy "b";copy "c"]) "a" = "a"

let disj_union_lenses (ls:lens List.t where
                          disjoint_regexps (stypes ls) &&
                          disjoint_regexps (vtypes ls))
  : (lens in union_regexps (stypes ls)
        <-> union_regexps (vtypes ls))
  = List.fold_left{lens}{lens}
    (fun (l_acc:lens) (l:lens) ->
       (l_acc | l))
    (copy EMPTY) ls

test disj_union_lenses #{lens}[copy [a]; "a" <-> "b"] = error
test get (disj_union_lenses #{lens}[copy "a";copy "b";copy "c"]) "a" = "a"
```

5.2.8 Lenses with List Arguments

These final two combinators take lists as arguments (and so have to be defined here instead of `Core`.)

First, we have `permute`.

`permute` The lens `permute` is an n-ary, permuting concatenation operator on lenses. Given a concrete string, it splits it into n pieces, applies the *get* function of the corresponding lens to each piece, reorders the abstract strings according to the fixed permutation specified by `sigma`, and concatenates the results. Given a permutation `sigma` and a list of lenses `l = #{lens}[l1;l2;...;ln]` with types `Ci <-> Ai`. It produces a lens with type `C1.C2...Cn <-> Asigma(1).Asigma(2)...Asigma(n).`

```
let lens_permute
  (sigma:int List.t)
  (ls:lens List.t where
    (List.valid_permutation{lens} sigma ls) &&
    ((concatable (stypes ls)) &&
     (concatable (List.permute{regexp} sigma (vtypes ls)))))
  : (lens in concat_regexps (stypes ls) <->
          concat_regexps (List.permute{regexp} sigma (vtypes ls)))
```

```
test stype (lens_permute #{int}[1;0] #{lens}["abc";"def"]) = "abcdef"
test vtype (lens_permute #{int}[1;0] #{lens}["abc";"def"]) = "defabc"
test get (lens_permute #{int}[2;0;1] #{lens}["abc";"def";"ghi"])
  "abcdefghi" = "defghiabc"
test get (lens_permute
            #{int}[2;1;0]
            #{lens}[(copy UALPHACHAR);
                    (copy UALPHACHAR);
                    (copy UALPHACHAR)]) "ABC" = "CBA"
```

`sortable, sort` The canonizer `sort` puts substrings into sorted order according to a list of regular expressions. An exception is raised if the unsorted string does not have exactly one substring belonging to each regular expression. This allows us to assign sort a type that is compact (though imprecise); see **?** for a discussion.

```
let sortable (rl:regexp List.t) : bool =
     disjoint_regexps (List.map{regexp}{regexp} (fun (r:regexp) -> r - EPSILON)
  && iterable_cex ((union_regexps rl) - EPSILON)

let sort
  (rl:aregexp List.t where sortable (List.map{aregexp}{regexp} rxdrop rl))
: (cn:canonizer where (uncanonized_type cn = (union_regexps (List.map{aregexp}{r
                   && (canonized_type cn = concat_regexps (List.map{aregexp}{reg

test canonize (sort #{regexp}[UALPHACHAR; DIGIT]) "A1" = "A1"
test canonize (sort #{regexp}[UALPHACHAR; DIGIT]) "1A" = "A1"
test canonize (sort #{regexp}[UALPHACHAR; DIGIT]) "A" = error
test sort #{regexp}["a";"a"] = error

test uncanonized_type (sort #{regexp}[UALPHACHAR; DIGIT]) =
  (UALPHACHAR | DIGIT)*

test canonized_type (sort #{regexp}[UALPHACHAR; DIGIT]) =
  (UALPHACHAR . DIGIT)
```

5.2.9 Miscellaneous

`iterate` The operator `iterate` compose `f` with itself `i` times using `b` for the first argument, i.e., `f(f(...(f b)...))` where `f` appears `i` times.

```
let iterate ('a) (i:int where (bgeq i 0)) (f:'a -> 'a) (b:'a) : 'a =
  List.fold_left{int}{'a}
    (fun (acc:'a) (i:int) -> f acc)
    b
    (List.mk_seq i)

test (iterate{regexp} 3 (fun (x:regexp) -> x | "(".x.")") [a-z]).get "((b))" = "
test (iterate{regexp} 3 (fun (x:regexp) -> x | "(".x.")") [a-z]).get "((((b))))"
```

`show` Gives a string representation of the value. Some values cannot be translated in full, e.g., functions.

```
let show : forall 'a => 'a -> string
```

5.3 Lists

The `List` module defines a datatype for polymorphic list structures. In this module we cannot use the Boomerang notation for lists because it is resolved using List.Nil and List.Cons, which are not valid names while the List module is being defined.

`'a t` A list is either the `Nil` list or a `Cons` of a head and a tail.

```
type 'a t = Nil | Cons of 'a * 'a t
```

`empty,nonempty` Predicates for detecting (non)empty lists.

```
let empty ('a) (l:'a t) : bool =
  match l with
    | Nil     -> true
    | Cons(_) -> false

let nonempty ('a) (l:'a t) : bool = not (empty{'a} l)
```

`hd,tl` The selectors `hd` and `tl` pull out the first and last parts of a `Cons`-cell, respectively.

```
let hd ('a) (xs:'a t) : 'a =
  let (Cons(x,_)) = xs in
  x

let tl ('a) (xs:'a t) : 'a t =
  let (Cons(_,xs)) = xs in
  xs
```

`fold_left` Boomerang does not support recursion. However, we provide the `fold_left` function on lists via a built-in primitive.

```
let fold_left ('a) ('b) (f:'b -> 'a -> 'b) (acc:'b) (l:'a t) : 'b
```

`length` Calculates the length of a list.

```
let length ('a) (l : 'a t) : int =
  fold_left{'a}{int}
    (fun (n:int) (v:'a) -> plus n 1)
    0 l

test length{bool} Nil{bool} = 0
test length{bool} (Cons{bool}(true,Cons{bool}(false,Nil{bool}))) = 2
```

`mk_seq` The function `mk_seq` returns a list of integers from `0` to `n-1`.

```
let mk_seq (n:int where n geq 0) : (l:int t where length{int} l = n)
```

`reverse` The function `reverse` can be defined straightforwardly using `fold_left`.

```
let reverse ('a) (l : 'a t) : 'a t =
  fold_left{'a}{'a t}
    (fun (t:'a t) (h:'a) -> Cons{'a}(h,t))
    Nil{'a}
    l
```

`append` The function `append` can be defined using `fold_left` and `reverse`.

```
let append ('a) (l1 : 'a t) (l2 : 'a t) : 'a t =
  fold_left{'a}{'a t}
    (fun (l:'a t) (x:'a) -> Cons{'a}(x,l))
    l2
    (reverse{'a} l1)
```

`map, rev_map` The function `map` can be defined (inefficiently) using `fold_left` and `reverse`. The `rev_map` is more efficient, but leaves the list reversed.

```
let rev_map ('a) ('b) (f:'a -> 'b) (l:'a t) : 'b t =
  fold_left{'a}{'b t}
    (fun (t:'b t) (h:'a) -> Cons{'b}(f h,t))
    Nil{'b}
    l

let map ('a) ('b) (f:'a -> 'b) (l:'a t) : 'b t =
  reverse{'b} (rev_map{'a}{'b} f l)
```

exists The function exists tests if a predicate holds of some element of the list.

```
let exists ('a) (t:'a -> bool) (l:'a t) : bool =
  fold_left {'a}{bool} (fun (b:bool) (h:'a) -> b || t h)
  false
  l
```

for_all The function for_all tests if a predicate holds of every element of the list.

```
let for_all ('a) (t:'a -> bool) (l:'a t) : bool =
  fold_left {'a}{bool} (fun (b:bool) (h:'a) -> b && t h)
  true
  l
```

member The function member tests if an element is a member of the list. It is defined using exists.

```
let member ('a) (x:'a) (l:'a t) : bool =
 exists{'a} (fun (h:'a) -> x = h) l
```

5.3.1 Permutations

A permutation is an integer list, mapping positions to other positions: if the `i`th entry of a permutation is the number `j`, then the `i`th element in the original list will be the `j`th element in the permuted list. A permutation for the list `#{bool}[true;true;false;true;false]` might be `#{int}[0;1;2;3;4]` (the identity permutation) or `#{int}[4;3;2;1;0]` (reversal).

valid_permutation The predicate valid_permutation is true when given the given permutation can be applied to the given list.

```
let valid_permutation ('a) (sigma:int t) (l:'a t) : bool

test valid_permutation{bool} Nil{int} Nil{bool} = true
test valid_permutation{bool}
 (Cons{int}(1,Cons{int}(0,Nil{int})))
 (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = true
test valid_permutation{bool}
  (Cons{int}(1,Cons{int}(1,Nil{int})))
  (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = false
test valid_permutation{bool}
  (Cons{int}(0 - 1,Cons{int}(1,Nil{int})))
  (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = false
test valid_permutation{bool}
```

```
    (Cons{int}(0,Cons{int}(1,Cons{int}(2,Nil{int}))))
    (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = false
  test valid_permutation{bool}
    (Cons{int}(1,Nil{int}))
    (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = false
  test valid_permutation{bool}
    (Cons{int}(1,Cons{int}(2,Nil{int})))
    (Cons{bool}(false,Cons{bool}(true,Nil{bool}))) = false
```

`permute` The operator `permute` permutes a list according to a given permutation.

```
  let permute ('a)
          (sigma:int t)
          (l:'a t where valid_permutation{'a} sigma l)
    : 'a t

  test permute{bool} Nil{int} Nil{bool} = Nil{bool}
  test permute{bool}
    (Cons{int}(0,Cons{int}(1,Nil{int})))
    (Cons{bool}(false,Cons{bool}(true,Nil{bool})))
  = (Cons{bool}(false,Cons{bool}(true,Nil{bool})))
  test permute{bool}
    (Cons{int}(1,Cons{int}(0,Nil{int})))
    (Cons{bool}(false,Cons{bool}(true,Nil{bool})))
  = (Cons{bool}(true,Cons{bool}(false,Nil{bool})))
  test permute{string}
    (Cons{int}(0,Cons{int}(2,Cons{int}(1,Nil{int}))))
    (Cons{string}("a",Cons{string}("b",Cons{string}("c",Nil{string}))))
  = (Cons{string}("a",Cons{string}("c",Cons{string}("b",Nil{string}))))
```

`permutations` The operator `permutations` returns a list of all possible permutations for lists of a given length.

```
  let permutations : (n:int where (n geq 0)) -> (int t) t

  test permutations 0 = (Cons{int t}(Nil{int},
                          Nil{int t}))
  test permutations 1 = (Cons{int t}(Cons{int}(0,Nil{int}),
                          Nil{int t}))
  test permutations 2 = (Cons{int t}(Cons{int}(0,Cons{int}(1,Nil{int})),
                          Cons{int t}(Cons{int}(1,Cons{int}(0,Nil{int})),
                          Nil{int t})))
```

`invert_permutation` The operator `invert_permutation` inverts a permutation `sigma`, calculating the permutation `sigma_inv` such that `permute_list{'a} sigma_inv (permua` for all `l`.

```
let invert_permutation : int t -> int t

let sort : forall 'a => ('a -> 'a -> int) -> 'a t -> 'a t

test sort{int} minus (Cons{int}(3, Cons{int}(4, Cons{int}(1, Cons{int}(0,
     Nil{int}))))) = (Cons{int}(0, Cons{int}(1, Cons{int}(3, Cons{int}(4,
     Nil{int})))))
```

5.4 Sorting

The `Sort` module defines functions for building lenses that do sorting.

5.4.1 Permutation Sorting

Using the `lens_permute` operator, and the functions for manipulating integer lists representing permutations from the `List` module it is straightforward to define lenses that do sorting.

`perms_regexps` The `perms_regexps` function computes the permutations of a list of regular expressions as a list of lists of regular expressions.

```
let perms_regexps (rl:regexp List.t) : (regexp List.t) List.t =
  List.map{int List.t}{regexp List.t}
    (fun (sigmai:int List.t) -> List.permute{regexp} sigmai rl)
    (List.permutations (List.length{regexp} rl))

test perms_regexps #{regexp}["a";"b"]
= #{regexp List.t}[#{regexp}["a";"b"]; #{regexp}["b";"a"]]
```

`perm_regexps` The `perm_regexps` function is similar but flattens the inner lists using `regexp_concat`.

```
let perm_regexps (rl:regexp List.t) : regexp List.t =
  List.map{regexp List.t}{regexp} concat_regexps (perms_regexps rl)

test perm_regexps #{regexp}["a";"b"]
= #{regexp}["ab";"ba"]
```

`perm_sortable` The `perm_sortable` predicate returns `true` iff the concatenations of all permutations of a list of regular expressions are unambiguous and also disjoint.

```
let perm_sortable (rl:regexp List.t) : bool =
  let perms = perms_regexps rl in
  List.for_all{regexp List.t} (fun (pi:regexp List.t) -> concatable pi) perms
  && disjoint_regexps (List.map{regexp List.t}{regexp} concat_regexps perms)
```

`perm_sort` The `perm_sort` lens sorts a list of regular expressions using instances of the `lens_permute` combinator.

```
let perm_sort
   (rl:regexp List.t where perm_sortable rl)
: (lens in union_regexps (perm_regexps rl) <-> concat_regexps rl) =
  let k : int = List.length{lens} rl in
  let ls_perms : lens List.t =
    List.map{int List.t}{lens}
      (fun (sigma:int List.t) ->
         let sigma_inv = List.invert_permutation sigma in
         lens_permute sigma (List.permute{lens} sigma_inv rl))
      (List.permutations (List.length{lens} rl)) in
  List.fold_left{lens}{lens}
    (fun (acc:lens) (permi:lens) -> acc || permi)
    (copy EMPTY) ls_perms

let l3 : (lens in ("abc" | "acb" | "bac" | "bca" | "cab" | "cba") <-> "abc") =
  perm_sort #{regexp}["a";"b";"c"]
test l3.get "abc" = "abc"
test l3.get "acb" = "abc"
test l3.get "bac" = "abc"
```

`perm_sort_concat` The `perm_sort_concat` quotient lens uses a canonizer built using `perm_sort` to sort the source string before it is processed by (the concatenation of) a list of lenses.

```
let perm_sort_concat
    (ls:lens List.t where perm_sortable (stypes ls) && concatable (vtypes ls))
  : (lens in union_regexps (perm_regexps (stypes ls)) <-> concat_regexps (vtypes
  = left_quot (canonizer_of_lens (perm_sort (stypes ls))) (concat_lenses ls)
```

`perm_concat` The `perm_sort_concat` lens does sorting, but its type on the source side grows as the *factorial* of the regular expressions being sorted. The final `sort_concat` lens uses the `sort` canonizer, which has a much more compact (although imprecise) type.

```
let sort_concat
   (ls:lens List.t where sortable (stypes ls) && concatable (vtypes ls))
 : (lens in (union_regexps (stypes ls))* <-> concat_regexps (vtypes ls))
 = left_quot (sort (astypes ls)) (concat_lenses ls)

let ls : (lens in [abc]* <-> "abc") =
  sort_concat #{lens}["a"; "b"; "c"]

test get ls "abc" = "abc"
test get ls "cba" = "abc"
test get ls "bca" = "abc"
test get ls "bba" = error
test get ls "dba" = error
test put ls "abc" "cba" = "abc"

let partition_sort_concat
  (ls:lens List.t where concatable (vtypes ls))
  (l:lens where sortable (List.Cons{regexp} (stype l,stypes ls))
           && splittable_cex (concat_regexps (vtypes ls)) (vtype l)* )
: (lens in (union_regexps (stypes ls) | (stype l))* <->
          (concat_regexps (vtypes ls) . (vtype l)* ))
 =
 let cn_partition =
   canonizer_of_lens (partition ((union_regexps (stypes ls)) - EPSILON) (stype l
 in
 let cn_sort = sort (stypes ls) in
 left_quot cn_partition (left_quot cn_sort (concat_lenses ls) . l* )

test (partition_sort_concat #{lens}[copy [A-Z]; copy [0-9]] (copy [a-z])).get "a
test (partition_sort_concat #{lens}[copy [A-Z]; copy [0-9]] (copy [a-z])).put "Z
```

5.5 Command line parsing

```
let get_prog_name : unit -> string
```

`create_bool` create_bool name default doc fulldoc creates a preference such that if -name is present in the command line, then the value will be true. If -name=false is present in the command line then the value will be false.

```
let create_bool (name:string) (default:bool) (doc:string) : bool_prefs

let alias_bool : bool_prefs -> string -> unit

let read_bool : bool_prefs -> bool
```

```
let create_int (name:string) (default:int) (doc:string) : int_prefs
let alias_int : int_prefs -> string -> unit
let read_int : int_prefs -> int
let create_string (name:string) (default:string) (doc:string) : string_prefs
let alias_string : string_prefs -> string -> unit
let read_string : string_prefs -> string
let create_string_list (name:string) (doc:string) : string_list_prefs
let alias_string_list : string_list_prefs -> string -> unit
let read_string_list : string_list_prefs -> string List.t
let print_usage : string -> unit
```

`extern_rest` `extern_rest ()` returns the preference for anonymous arguments.

```
let extern_rest : unit -> string_list_prefs
let extern_output : unit -> string_prefs
let extern_lens : unit -> string_list_prefs
let extern_source : unit -> string_list_prefs
let extern_view : unit -> string_list_prefs
let extern_expression : unit -> string_list_prefs
let extern_check : unit -> string_list_prefs
let extern_include : unit -> string_list_prefs
let extern_test : unit -> string_list_prefs
let extern_testall : unit -> bool_prefs
let extern_debug : unit -> string_list_prefs
let extern_debugtimes : unit -> bool_prefs
let extern_log : unit -> bool_prefs
let extern_logfile : unit -> string_prefs
let extern_terse : unit -> bool_prefs
let extern_timers : unit -> bool_prefs
let extern_colorize : unit -> bool_prefs
```

5.6 System functions

`read` The `read` function reads the contents of a file from the local filesystem. If the argument is -, it reads the contents of the standard input.

```
let read : string -> string
```

`write` The `write` function writes a string to a file on the local filesystem. If the name of the file is -, the output is the standard output.

```
let write : string -> string -> unit
let put_str : string -> unit
= write "-"
```

`exec` The `exec` function executes a shell command.

```
let exec : string -> string
```

`file_exists` Test if a file with the given name exists.

```
let file_exists : string -> bool
```

`is_directory` Returns `true` if the given name refers to a directory, `false` if not.

```
let is_directory : string -> bool
```

`remove` Remove the given file name from the file system.

```
let remove : string -> unit
```

`rename` Rename a file. The first argument is the old name and the second is the new name. Returns `true` iff the file has been renamed.

```
let rename : string -> string -> bool
```

`getcwd` Return the current working directory of the process.

```
let getcwd : unit -> string
```

`os_type` Operating system currently executing Boomerang. The return is the same as the ocaml function `sys.os_stype` and is

- `"Unix"` (for all Unix versions, including Linux and Mac OS X),
- `"Win32"` (for MS-Windows, OCaml compiled with MSVC++ or Mingw),
- `"Cygwin"` (for MS-Windows, OCaml compiled with Cygwin).

```
let os_type : string
```

Chapter 6

The Boomerang System

6.1 Running Boomerang

All of the interactions with Boomerang we have seen so far have gone via unit tests. This works well for interactive lens development, but is less useful for batch processing of files. Boomerang can also be involved from the command line:

```
Usage:
    boomerang [get] l S           [options]     : get
  or boomerang [put] l V S        [options]     : put
  or boomerang create l V         [options]     : create
  or boomerang M.boom [N.boom...] [options]     : run unit tests for M, N, ...
```

To try this out, create a file `comps-conc.txt` containing the following lines:

```
Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, American
Benjamin Britten, 1913-1976, English
```

and run the command

```
boomerang get QuickStart.comps comps-conc.txt
```

You should see

```
Jean Sibelius, Finnish
Aaron Copland, American
Benjamin Britten, English
```

written to the terminal.

Now let's do the same thing, but save the results to a file:

```
boomerang get QuickStart.comps_cmdline comps-conc.txt -o comps-abs.txt
```

Next let's edit the abstract file to

```
Jean Sibelius, Finnish
Benjamin Britten, English
Alexandre Tansman, Polish
```

and *put* the results back:

```
boomerang put QuickStart.comps_cmdline comps-abs.txt comps-conc.txt
```

You should see

```
Jean Sibelius, 1865-1957, Finnish
Benjamin Britten, 1913-1976, English
Alexandre Tansman, 0000-0000, Polish
```

printed to the terminal.

6.2 Running a Boomerang program

When Boomerang is called with another name, Boomerang run the module with this name passing all command line arguments to the module. Boomerang still interpret all arguments that are not interpreted by the module.

The `examples/address.boom` is a complete Boomerang program. It does transformations between VCard, XCard and CSV files. To run it, you need to create a link to Boomerang with the name `address`:

```
> ln -s /path/to/trunk/bin/boomerang address
```

If you run `address` now Boomerang will call the Address module. To try this out, create a file `contacts.csv` containing the following lines:

```
Doe, John, hello world (note), 792-8134 (h), 732-4684 (h)
```

and run the command

```
./address get contacts.csv xml
```

You should see

```
<xcard>
  <vcard>
    <n>
      <family>Doe</family>
      <given>John</given>
    </n>
    <note>hello world</note>
    <tel-home>792-8134</tel-home>
    <tel-home>732-4684</tel-home>
  </vcard>
</xcard>
```

written to the terminal. As address is not only one lens between two types, we need to specify to which format we are converting (the *xml* in our previous example). For example, to transform the csv into a vcard you should run

```
./address get contacts.csv vcf
```

The *put* is similar, but both arguments are a file and first is as the updated view and the second is the old source. To try this out, create a file `updated.xml` containing the following lines:

```
<xcard>
  <vcard>
    <n><family>Doe</family><given>Sally</given></n>
    <tel-home>792-8134</tel-home>
    <tel-home>732-4684</tel-home>
  </vcard>
  <vcard>
    <n>
      <family>Doe</family>
      <given>John</given>
    </n>
    <note>updated</note>
    <tel-home>792-8134</tel-home>
  </vcard>
</xcard>
```

and run the command

```
./address put updated.xml contacts.csv
```

You should see

```
Doe, Sally, 792-8134 (h), 732-4684 (h)
Doe, John, updated (note), 792-8134 (h)
```

6.3 Creating a Boomerang program

All you need to create a Boomerang program, in addition to write a lens, is to write a `main` function that takes `unit` and returns `unit` or `int`. In the second case, the return of `main` is the return code of the program.

Boomerang programas receive command line arguments using the `Prefs` library. The `bibtex.boom` example can be a good start to see how to write a Boomerang program, just look at the main function at the end of the file.

If you need to do more than just use anonymous arguments, see the `Prefs` library and the `conflin.boom` example.

6.4 Navigating the Distribution

If you want to check out the code, here is one reasonable order to look at the files:

`src/lenses/core.boom`	core lenses
`src/lenses/prelude.boom`	important derived lenses
`src/blenses.ml`	native definitions of lenses and canonizers
`src/bcompiler.ml`	the Boomerang interpreter
`src/balign.ml`	the alignment functions
`src/toplevel.ml`	the top-level program

Chapter 7

Case Studies

Under construction. For now, see the demos in the `examples` directory.

In the `examples` directory, you can find some of the other Boomerang programs we have written:

- `demo.boom`: A simple demo, similar to composers lens.
- `addresses.boom` : VCard, CSV, and XML-formatted address books.
- `bibtex.boom`: BiBTeX and RIS-formatted bibliographies.
- `uniProtV2.boom`: UniProtKB / SwissProt lens.
- `conflin.boom`: Management tool for multiple versions of a file.
- `xsugar/*`: example transformations from the XSugar project.

We will continue adding to this set of examples as we tidy and package our code... and we hope you'll write and let us know about the lenses you write!

Bibliography

Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. Technical Report MS-CIS-05-27, Dept. of Computer and Information Science, University of Pennsylvania, December 2005.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29 (3):17, 2007. ISSN 0164-0925. Extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, British Columbia*, September 2008.