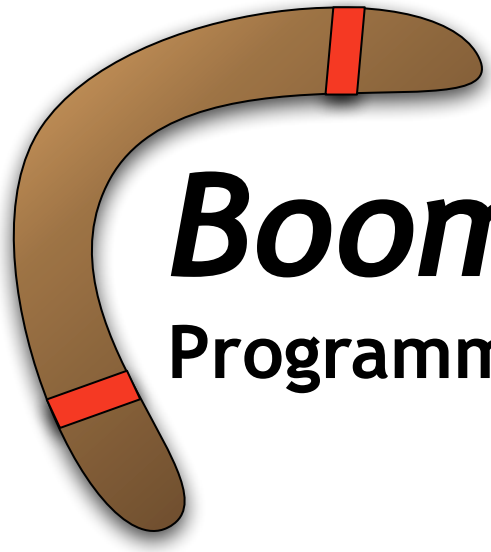

# Boomerang
## Programmer's Manual

J. Nathan Foster and Benjamin C. Pierce

with

Aaron Bohannon, Michael Greenberg,
Alan Schmitt, and Alexandre Pilkiewicz

July 25, 2008

## 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

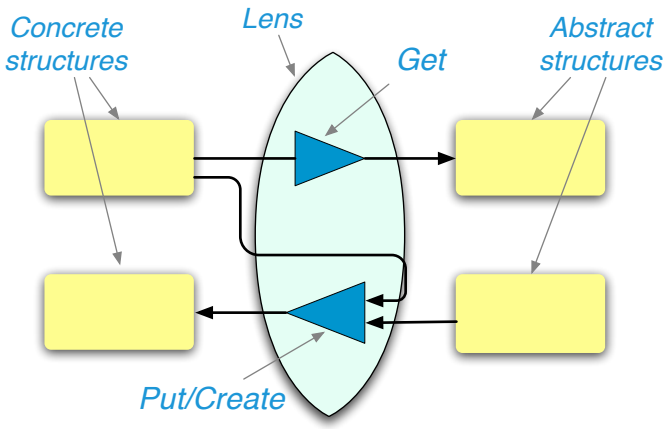


Figure 1.1: Lens Terminology

$C$ of "concrete" strings and a set $A$ of "abstract" ones has three components:

$$\begin{aligned} l.get &\in C \longrightarrow A \\ l.put &\in A \longrightarrow C \longrightarrow C \\ l.create &\in A \longrightarrow C \end{aligned}$$

*get* is the forward transformation and is a total function from $C$ to $A$. The backwards transformation comes in two flavors. The first, *put*, takes two arguments, a modified $A$ and an old $C$, and produces an updated $C$. The second, *create*, handles the special case where we need to compute a $C$ from an $A$ but have no $C$ to use as the "old value". It fills in any information in $C$ 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 $c \in C$ and $a \in A$:

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

$$l.get\ (l.put\ a\ c) = a \qquad \text{(PutGet)}$$

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

The first law requires that if *put* is invoked with an abstract string that is identical to the string obtained by applying *get* to the old concrete string—i.e., if the edit to the abstract string is a no-op—then it must produce the same concrete string. The second and third laws state that *put* and *create* must propagate all of the information in their $A$ arguments to the $C$ 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 such as strings, many applications require that lenses match up corresponding pieces of the concrete and abstract strings. Boomerang includes combinators for describing how data should be aligned using natural notions of "chunk" and "keys". We call lenses that use these features *dictionary lenses*.

- 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 primtives 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 `<composers>`. 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 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. After that, we suggest exploring the `examples` directory, which contains some of the larger demos we've built, and consulting the rest of this manual as needed. 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]). 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 boomerang/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 Boomerant 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 3-32: 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
copy built-in: type errors in
  [Hello World]
<<HERE>>
  [!!]
```

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 c : 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 a : string =
Jean Sibelius, Finnish
Aaron Copland, American
Benjamin Britten, English
```

### 2.3.1 Basic Composers Lens

The lens that maps—bidirectionally—betweeen 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 c = a
test comps.put a into c = c
```

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 Dictionary 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" (`<comp>`) and 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 = "" | <comp> . (newline . <comp>)*
```

restores lines using the name on each line as a key, rather than by position. For the details of how this all works, see Bohannon et al. [2008]. 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.

# Chapter 3

# The Boomerang Language

The Boomerang language provides convienent 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.

## 3.1 Lexing

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

### 3.1.1 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.

### 3.1.2 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
.get    .put          .create  .canonize       .choose
.ctype  .domain_type  .atype   .codomain_type  .bij
```

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`.

### 3.1.3 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.

String literals enclosed in `/` and `/` are lexed as singleton regular expressions.

## 3.2 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).

### 3.2.1 Modules and Declarations

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

⟨*Decl*⟩ ::= '`module`' ⟨*LIdent*⟩ '=' ⟨*Decl*⟩* '`end`'
| '`type`' ⟨*TyVarList*⟩ ⟨*LIdent*⟩ '=' ⟨*DTSortList*⟩
| '`let`' ⟨*Id*⟩ (⟨*Param*⟩)+ ('`:`' ⟨*Sort*⟩)? '=' ⟨*Exp*⟩
| '`let`' ⟨*PairPat*⟩ (⟨*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 **??**).

⟨*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. Finally, unit tests of the form `test` $e$ `:` $s$ and `test` $e$ `: ?` test the sort of $e$ rather than its value.

### 3.2.2 Expressions

⟨*Exp*⟩ ::= '`let`' ⟨*Id*⟩ (⟨*Param*⟩)+ ('`:`' ⟨*Sort*⟩)? '`=`' ⟨*Exp*⟩ '`in`' ⟨*Exp*⟩
  | '`let`' ⟨*PairPat*⟩ ('`:`' ⟨*Sort*⟩)? '`=`' ⟨*Exp*⟩ '`in`' ⟨*Exp*⟩
  | '`fun`' (⟨*Param*⟩)+ ('`:`' ⟨*Sort*⟩)? '`->`' ⟨*Exp*⟩
  | ⟨*CaseExp*⟩

⟨*CaseExp*⟩ ::= '`match`' ⟨*Exp*⟩ '`with`' ⟨*BranchList*⟩ '`:`' ⟨*Sort*⟩
  | '`(`' '`match`' ⟨*Exp*⟩ '`with`' ⟨*BranchList*⟩ '`)`' '`:`' ⟨*Sort*⟩
  | '`begin`' '`match`' ⟨*Exp*⟩ '`with`' ⟨*BranchList*⟩ '`end`' '`:`' ⟨*Sort*⟩
  | ⟨*ComposeExp*⟩

**Parameters**

⟨*Param*⟩ ::= '`(`' ⟨*Id*⟩ '`:`' ⟨*Sort*⟩ '`)`'
  | '`(`' ⟨*TyVarIdent*⟩ '`)`'
  | ⟨*TyVarIdent*⟩

**Branches**

⟨*Branch*⟩ ::= ⟨*Pat*⟩ '->' ⟨*InfixExp*⟩

⟨*BranchList*⟩ ::= ('|')? ⟨*Branch*⟩ ('|' ⟨*Branch*⟩)*

⟨*ComposeExp*⟩ ::= ⟨*ComposeExp*⟩ ';' ⟨*BarExp*⟩
| ⟨*BarExp*⟩

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

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

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

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

⟨*CommaExp*⟩ ::= ⟨*CommaExp*⟩ ',' ⟨*InfixExp*⟩
| ⟨*InfixExp*⟩

⟨*InfixExp*⟩ ::= ⟨*DotExp*⟩
| ⟨*TildeExp*⟩
| ⟨*AmpExp*⟩
| ⟨*LensComponentExp*⟩
| ⟨*AppExp*⟩ '-' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '&&' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '<->' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '<=>' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '<' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '<=' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '>' ⟨*AppExp*⟩
| ⟨*AppExp*⟩ '>=' ⟨*AppExp*⟩
| ⟨*AppExp*⟩

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

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

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

⟨*LensComponentExp*⟩ ::= ⟨*AppExp*⟩ ('`.get`' ⟨*AppExp*⟩
  | ⟨*AppExp*⟩ ('`.put`' ⟨*AppExp*⟩ '`into`' ⟨*AppExp*⟩
  | ⟨*AppExp*⟩ ('`.create`' ⟨*AppExp*⟩
  | ⟨*AppExp*⟩ ('`.canonize`' ⟨*AppExp*⟩
  | ⟨*AppExp*⟩ ('`.choose`' ⟨*AppExp*⟩
  | ⟨*AppExp*⟩ ('`.ctype`'
  | ⟨*AppExp*⟩ ('`.atype`'
  | ⟨*AppExp*⟩ ('`.domain_type`'
  | ⟨*AppExp*⟩ ('`.codomain_type`'
  | ⟨*AppExp*⟩ ('`.bij`'

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

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

⟨*TyExp*⟩ ::= ⟨*TyExp*⟩ '`{`' ⟨*Sort*⟩ '`}`'
  | ⟨*TyExp*⟩

⟨*AExp*⟩ ::= '`(`' ⟨*Exp*⟩ '`)`'
  | '`begin`' ⟨*Exp*⟩ '`end`'
  | ⟨*Qid*⟩
  | ⟨*MatchExp*⟩
  | '`#`' '`{`' ⟨*SortList*⟩ '`}`' ⟨*List*⟩
  | ⟨*Character*⟩
  | ⟨*Integer*⟩
  | ⟨*Boolean*⟩
  | ⟨*CharSet*⟩
  | ⟨*NegCharSet*⟩
  | ⟨*String*⟩
  | ⟨*RegExpString*⟩
  | '`()`'

⟨*MatchExp*⟩ ::= <⟨*Qid*⟩ >
  | <⟨*LIdent*⟩ '`:`' ⟨*Qid*⟩ >
  | <'`~`' ⟨*Qid*⟩ >
  | <'`~`' ⟨*LIdent*⟩ '`:`' ⟨*Qid*⟩ >
  | <'`~`' '`{`' ⟨*Float*⟩ '`}`' ⟨*Qid*⟩ >
  | <'`~`' '`{`' ⟨*Float*⟩ '`}`' ⟨*LIdent*⟩ '`:`' ⟨*Qid*⟩ >

### 3.2.3 Repetitions

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

### 3.2.4 Lists

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

### 3.2.5 Patterns

⟨*Pat*⟩ ::= ⟨*UIdent*⟩ ⟨*PairPat*⟩
  | ⟨*QualIdent*⟩ ⟨*PairPat*⟩
  | ⟨*PairPat*⟩

⟨*PairPat*⟩ ::= ⟨*PairPat*⟩ '`,`' ⟨*APat*⟩
  | ⟨*APat*⟩

⟨*APat*⟩ ::= '`_`'
  | ⟨*LIdent*⟩
  | '`()`'
  | ⟨*Integer*⟩
  | ⟨*Boolean*⟩
  | ⟨*String*⟩
  | ⟨*UIdent*⟩
  | ⟨*Qident*⟩
  | '`(`' ⟨*Pat*⟩ '`)`'

### 3.2.6 Sorts

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

⟨*ArrowSort*⟩ ::= ⟨*ProductSort*⟩ '`->`' ⟨*ArrowSort*⟩
  | ⟨*ProductSort*⟩

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

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

⟨*BSort*⟩ ::= '(' ⟨*Sort*⟩ ')'
  | ⟨*ASort*⟩

⟨*BSort*⟩ ::= ⟨*QVar*⟩
  | 'char'
  | 'string'
  | 'regexp'
  | 'lens'
  | 'int'
  | 'bool'
  | 'canonizer'
  | 'unit'
  | ⟨*TyVar*⟩

⟨*TyVar*⟩ ::= ⟨*TyVarIdent*⟩

⟨*TyVarList*⟩ ::= ⟨*TyVar*⟩
  | '(' ⟨*TyVar*⟩ (',' ⟨*TyVar*⟩)* ')'

⟨*DTSort*⟩ ::= ⟨*UIdent*⟩
  | ⟨*UIdent*⟩ 'of' ⟨*Sort*⟩

⟨*DTSortList*⟩ ::= ⟨*DTSort*⟩ ('|' ⟨*DTSort*⟩)*

### 3.2.7 Identifiers

⟨*Id*⟩ ::= ⟨*LIdent*⟩
  | ⟨*UIdent*⟩

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

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

# Chapter 4

# 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 3.2.1.

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 primtitives are "well behaved,". However, for getting up to speed with Boomerang programming, the shorter (conference) versions should suffice.

## 4.1 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`. The reason that we do this, rather than resolving them directly to the primitive values, is that it facilitates checking the preconditions on primtive values using dependent refinement types.

Every value defined in `Core` is available by default in every Boomerang program.

### 4.1.1 Equality

`equals` The polymorphic `equals` operator is partial: comparing function, lens, or canonizer values results in a run-time expception. The infix = operator desugars into `equals`.

```
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
```

### 4.1.2 Booleans

`land, lor, not` These operators are the standard functions on booleans. The infix operators `&&` and `||` resolve to `land` and `lor` respectively (when applied to booleans; `||` also resolves to `lens_union` when applied to lenses.)

```
let land : bool -> bool -> bool
let lor : bool -> bool -> bool
let land : bool -> bool -> bool
```

### 4.1.3 Integers

`string_of_int` The operator `string_of_int` converts an integer to a string in the obvious way.

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

`bgt,blt,bgeq,bleq` These operators are the standard comparisons on integers. Infix operators >, <, >=, <= resolve to these operators. In this module, we bind them to 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 bdiv : int -> int -> int
let bmod : int -> int -> int
```

### 4.1.4 Characters

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

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

### 4.1.5 Strings

`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
```

### 4.1.6 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. (Also, strings delimited by `/` in the lexer desugar into applications of `str`.)

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

`string_of_regexp` The `string_of_regexp` function a regular expression to a string.

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

`regexp_union` The `regexp_union` operator forms the union of two values of type `regexp`. 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 values of type `regexp`. 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 `regexp`. 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
```

`inter` The `inter` operator forms the intersection of two `regexp` values. The infix symbol `&` desugars into `inter`.

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

`diff` The `diff` operator forms the difference of two `regexp` values. The infix symbol `-` desugars into `diff`.

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

`shortest` The function `shortest` computes a representative of a regular expression whose length is minimal.

```
let shortest : regexp -> string
```

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

```
test shortest (regexp_iter [A-Z] 1 3) = "A"
test shortest [] = 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` The `equiv` function tests if two regular expressions denote the same language.

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

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

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

```
let matches : 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` The `disjoint` function tests whether two regular expressions denote disjoint languages. It is defined using `is_empty` and `inter`.

```
let disjoint (r1:regexp) (r2:regexp) : bool =
  is_empty (inter r1 r2)

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

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

```
let splittable : 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` The `iterable` function tests whether the iteration of a regular expression is ambiguous.

```
let iterable : 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 expresion `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 : regexp -> string -> int

test count [A-Z] "" = 0
test count [A-Z] "ABC" = 3
test count (regexp_iter [A-Z] 0 1) "ABC" = 3
test count (regexp_iter [A-Z] 0 1) "123" = 0
```

### 4.1.7 Lens Components

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

```
let get : lens -> string -> string
```

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

```
let put : lens -> string -> string -> string
```

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

```
let create : lens -> string -> string
```

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

```
let ctype : lens -> regexp
```

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

```
let atype : lens -> regexp
```

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

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

### 4.1.8 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) : lens

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 ctype (copy [A-Z]) = [A-Z]
test atype (copy [A-Z]) = ctype (copy [A-Z])
```

`const` The `const` lens takes as arguments a regular expression `R`, a string `u`, and a string `v`. Its *get* function is the constant function that returns `u`, its *put* function restores its concrete argument, and its *create* function returns the default string `v`.

```
let const : regexp -> string -> string -> lens

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

`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) : lens =
 const r s (shortest 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) : lens =
  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 : lens -> lens -> lens

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 : lens -> lens -> lens

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 : lens -> lens -> lens

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 : lens -> lens -> lens

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 : lens -> lens -> lens

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_iter` The `lens_iter` operator iterates a lens. The iterations of the concrete and abstract types of the lens must both be unambiguous. The overloaded operators `*`, `+`, `?`, `{m,n}` and `{n,}` all desugar into instances of `lens_iter` when applied to a lens. If the second argument is negative, then the iteration is unbounded. For example, `l*` desugars into `lens_iter l 0 (-1)`.

```
let lens_iter : lens -> int -> int -> lens

test get (lens_iter (copy [A-Z]) 0 4) "" = ""
test get (lens_iter (copy [A-Z]) 0 4) "ABCD" = "ABCD"
test put (lens_iter (copy [A-Z]) 0 4) "AB" "ABCD" = "AB"
test create (lens_iter (copy [A-Z]) 0 4) "A" = "A"
```

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

```
let invert : lens -> lens

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 : lens -> string -> lens

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

`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 : regexp -> regexp -> lens

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

### 4.1.9 Dictionary Lenses

The next few primitives construct lenses for handling ordered data called dictionary lenses. For details, see Bohannon et al. [2008].

`key` The `key` operator takes a regular expression as an argument. Its *get*, *put*, and *create* components are like `copy`, but its *key* component is the identity function on the abstract string.

```
let key : regexp -> lens
```

`dmatch` The `dmatch` operator takes as arguments a string `t` and a lens `l`. It builds a dictionary lens that applies `l` in the `t` "chunk". The type checker requires that the same lens be used for every instance of `dmatch` with the same tag. In Boomerang, we check this condition using a conservative approximation: we assign every lens an integer unique identifier when it is constructed and require that for each tag `t`, the lenses in each occurrence of `dmatch` with `t` have the same unique identifier.

```
let dmatch : string -> lens -> lens
```

`smatch` The `smatch` operator is like `dmatch` but uses a similarity-based lookup operator for chunks. The first string must represent a floating point number, which is used as the threshold.

```
let smatch : string -> string -> lens -> lens
```

`forgetkey` The `forgetkey` operator takes a lens `l` as an argument. It behaves like `l`, but overrides its *key* component with the constant function returning the empty string.

```
let forgetkey : lens -> lens
```

### 4.1.10 Canonizers

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

```
let canonize : canonizer -> string -> string
```

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

```
let choose : canonizer -> string -> string
```

`uncanonized_type` The `rtype` 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
```

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

```
let canonized_type : canonizer -> regexp
```

`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 : lens -> canonizer
```

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

```
let canonizer_concat : canonizer -> canonizer -> canonizer
```

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

```
let canonizer_union : canonizer -> canonizer -> canonizer
```

`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)`.

`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 : int -> regexp -> char -> string -> canonizer
```

The following unit test illustrates the *choose* component of `columnize` (we would normally write `[a-z ]*` instea of `(regexp_iter [a-z ] 0 (minus 0 1))` in any module other than `Core`.)

```
test choose (columnize 5 (regexp_iter [a-z ] 0 (minus 0 1)) ' ' "\n")
       "a b c d e f g" =
<<
  a b c
  d e f
  g
>>
```

### 4.1.11 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 : canonizer -> lens -> lens
```

```
test get
  (left_quot (columnize 5 (regexp_iter [a-z ] 0 (minus 0 1)) ' ' "\n")
             (copy (regexp_iter [a-z ] 0 (minus 0 1))))
<<
  a b c
  d e f
  g
>>
= "a b c d e f g"

test create
  (left_quot (columnize 5 (regexp_iter [a-z ] 0 (minus 0 1)) ' ' "\n")
             (copy (regexp_iter [a-z ] 0 (minus 0 1))))
"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 : lens -> canonizer -> lens
```

`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 : lens -> (string -> string) -> regexp -> lens

test get (dup1 (copy [A-Z]) (get (copy [A-Z])) [A-Z]) "A" = "AA"
test put (dup1 (copy [A-Z]) (get (copy [A-Z])) [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 : lens -> (string -> string) -> regexp -> lens

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

### 4.1.12 Miscellaneous

`read` The `read` function reads the contents of a file from the local filesystem.

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

`blame` The `blame` function (not shown) is used in the Boomerang intepreter to report dynamic contract failures.

### 4.1.13 Synchronization

`sync` The `sync` function takes a lens an three strings as arguments. It synchronizes three strings using a type-respecting synchronization algorithm based on diff3. (The synchronization type is extracted from the the lens argument.)

```
let sync : lens -> string -> string -> string ->
           (string * string * string * string)
```

## 4.2 The Standard Prelude

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

### 4.2.1 Regular Expressions

`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 = //
```

`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` 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`.

```
let containing (R:regexp) : regexp = ANY . R . ANY
```

`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]
```

### 4.2.2 Lenses

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

```
let ins (s:string) : lens = "" <-> 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 = R <-> ""

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

### 4.2.3 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) (c:string) : lens =
  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 =
 qconst C A (shortest A) (shortest 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` The lens `qins` is like `ins` but accepts a regular expression in the *put* direction. It is defined using right quotienting and `ins`.

```
let qins (E:regexp) (e:string) : lens =
  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") = ""
```

`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 =
  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
```

### 4.2.4 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
```

### 4.2.5 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
```

### 4.2.6 Lenses with List Arguments

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

`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.

```
let permute : int List.t -> lens List.t -> lens

test get (permute
            #{int}[2;1;0]
            #{lens}[(copy UALPHACHAR);
                    (copy UALPHACHAR);
                    (copy UALPHACHAR)]) "ABC" = "CBA"
```

`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 an extended discussion.

```
let sort : regexp List.t -> canonizer

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

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

## 4.3 Lists

The `List` module defines a datatype for fpolymorphic list structures.

`'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
```

`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
```

`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
```

`map` The function `map` can be defined (inefficiently) using `fold_left` and `reverse`.

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

`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
```

`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
```

# Chapter 5

# The Boomerang System

## 5.1 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 C          [options]     : get
 or boomerang [put] l A C        [options]     : put
 or boomerang create l A         [options]     : create
 or boomerang sync l O C A       [options]     : sync
 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.

## 5.2 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 |
| `examples/*` | many real-world lenses |
| `src/bcompiler.ml` | the Boomerang interpreter |
| `src/sync.ml` | a synchronization algorithm |
| `src/toplevel.ml` | the top-level program |

# Chapter 6

# Case Studies

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

In the `boomerang/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.
- `uniprot.boom`: UniProtKB / SwissProt lens.
- `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*, 2007. To appear. 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.