Introducing Grammars to Boomerang

Abstract
Most programs work in one direction: given an input, they compute an output. In a bidirectional language, however, programs can be operated in two modes: in the forward direction they map inputs to outputs, and in the reverse direction they map outputs back to inputs. These languages are useful for solving problems in a variety of applications where relationships between data need to be maintained (e.g. data converters and synchronizers, databases, graphical user interfaces, compilers, system administration, and software engineering.)

The goal of this project is to extend Boomerang, a general-purpose bidirectional language (http://www.seas.upenn.edu/~harmony/), to support linear right-recursive grammars. This entails designing a new syntax for describing bidirectional transformations that is both practical and elegant, and successfully integrating it with the Boomerang system. Boomerang is a programming language that produces lenses, which are well-behaved bidirectional transformations on textual data formats. Before this project, it had no support for producing lenses other than using its primitive combinators. Grammars make bidirectional transformations much more succinct, and consequently easier to read and write, make bidirectional programming for new users less painful.

Concretely, the project began by building a full-blown standalone compiler, which took a grammar as input and produced a lens. After completing this task, the compiler was redesigned and directly integrated with the Boomerang compiler (specifically its parser, typechecker, and interpreter). We based our syntax on an intuitive representation of lens transformations, with guidance from Benjamin Pierce and Nate Foster.

1. Related Work
Boomerang is a bidirectional programming language developed by Pierce et al. that has been used to build transformers for complex real-world data formats including the SwissProt genomic database. Basic lenses, or bidirectional programs, served as the starting point for their work. Formally, a basic lens maps between a set of inputs (“concrete structures”) and a set of outputs (“abstract structures”). When read from left to right, it produces an output from a given input. When read from right to left, however, this same program, by examining the lens definition, describes a “backwards” function that takes some modified output, along with the original input, to produce a modified input that would have produced the modified output in the “forwards” direction.

Lenses have three basic functions to handle these transformations, get, put, create. get refers to the forward transformation of an input to an output. The put functionality refers to taking a modified output, combined with the original input, to create an updated original input. The create component is like put except that it only takes an output argument; it supplies defaults for the information discarded by get in situations where only the output is available.
The simplest way to demonstrate this transformation (not using lenses) would be to write two functions and verify by hand that they fit together. In other words, verify that the output of one function, when mapped to the input of the other, yields the original input. In other words, verify that when these two functions are composed, they yield the identity function. Boomerang takes this idea, and combines these two functions into the lens methodology described above.

This image illustrates the view-update problem facing many of today’s applications. For example, a database requires a view definition to create the view. Changes made by a user in the view in turn need to be reflected in the database. To do this, it passes through an update translation policy, which depends on the view definition and the structures of the database and view. This can become tedious to maintain as the other components of the system change.

We can see by the following image how lenses make this problem easier to solve. There is no need to write both a view definition and an update translation. The lens handles both of these.

Examples of applications for Boomerang can be found within address books, tex documents, and XML conversions to name a few. The application to an address book is important because it aids data synchronization across different formats. For instance, Boomerang could map between CSV and an XML format, handling changes made in one, and representing those changes in the other. (Pierce et al., 2007)

Below we can see an example of a sample lens transformation.

<!-- Code not specified in the text --}

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

Jean Sibelius, 1865-1956
XSugar can be considered as a specialized bidirectional language, for succinctly describing conversions between XML and ASCII formats. XSugar is pretty simple syntactically and concisely accomplishes the task it was designed to do very well. It has a much smaller scope than Boomerang and is less robust. As a result of its less powerful nature, its syntax, for most programs, tends to be slightly less overbearing.

Below is an XSugar grammar for describing the composers transformation from above:

```
db : [comps cs] = <composers> [comps cs] </>
comps : [comp c] [comps cs] = [comp c] [comps cs] :
  comp : [Name n] ",” [Birth b] "-" [Death d] =
      <composer>
      <name> [Name n] </>
      <years birth=[Birth b] death=[Death d] />
      <nationality> [Nationality] </>
    </>
```

Given the growing benefits of using XML, having a tool like XSugar eliminates the need for two syntaxes that up until now, users were required to have. For example, before XSugar, to convert from XML to a non-XML format, one could use an XSLT stylesheet. On the other hand, in order to translate between non-XML and XML, a customized program in some high level language was required (Schwartzback et al., 2005). One of our tasks was to develop a compiler to compile XSugar down to Boomerang. This resulted in our deeper understanding of lenses, Boomerang, and transformations in general. Since Boomerang is more powerful than XSugar, seeing how a smaller language like XSugar is designed is very relevant to our goals. Becoming familiar with these bidirectional languages was important because when designing a robust syntax, knowledge of merely one bidirectional language would not be ideal. Essentially, we solved a subset of our larger problem first by engineering this compiler.

Nguyen et al. developed a generic language also centered around XML. This language defines iterators for XML data. This development was spurred by the sensitivity of XML to types. Even simple operations, which in XML are performed by iterators, can modify the types of documents. Things brings about a need for typing by polymorphism-like iterators. Traditionally, iterators are hard-coded and do not cover standard uses of XML. Thus, Nguyen et al. developed a small language of combinators, called filters. This allows the creation of XML transformations that accurately type the application (Nguyen et al., 2008).

A recent development given the advent of more bidirectional programming languages is the quotient lens. Quotient lenses are a slightly more refined version of the original lenses developed by Pierce et al, in that they are still used to describe similar bidirectional transformations. The idea is that all lenses do not need to be too strong in practice. There are some details like white space that can be modified after the transformation. The Quotient Lenses paper by Pierce, Foster, and Pilkiewicz explores the development of these quotient lenses, which are well behaved modulo equivalence relations controlled by the programmer. Additionally, it contains a section that specifically defines a grammar for XSugar productions, which was a great tool in beginning our first large objective, the XSugar to Boomerang compiler.

Our changes to the Boomerang compiler make drastic improvements to the language. XSugar is so simple that it can not be used beyond basic ASCII to XML transformations. With the addition of grammars (inspired by XSugar), users can harness the power of Boomerang, while being able to avoid a lot of the overhead and lower level syntax that Boomerang programming requires. Our grammars are more readable, succinct, and understandable to new and even experienced users. Given that Boomerang is a relatively new language and is preparing for a wide reaching and publicized release, the addition of this grammar functionality should enhance its appeal.
2. Technical Approach

Our system consists of a grammar syntax that is directly integrated into Boomerang. These new Boomerang programs can combine the existing functionality with the new grammar support. As a result, when writing a grammar, the lens representing that grammar’s transformation remains under the covers. This saves the user from having to write a complicated lens. As a result, Boomerang is more accessible to many developers as the barrier to implementing a Boomerang program is a significantly lower learning curve.

Consider this transformation:

Input:

```xml
<composer>
  <name>Jean Sibelius</name>
  <years birth=1865 death=1957/>
  <nationality> </nationality>
</composer>

<composer>
  <name>Aaron Copland</name>
  <years birth=1910 death=1990/>
  <nationality> </nationality>
</composer>

<composer>
  <name>Benjamin Britten</name>
  <years birth=1913 death=1976/>
  <nationality> </nationality>
</composer>
```

Output:

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

The Boomerang lens required to complete such a transformation would look like this. This also happens to be the most concise way of writing this Boomerang lens. It could look even less readable.

```plaintext
let comp : lens =
  lens_permute #{int}[0;1;2;3;4;5;6]
  #\{lens\}[["<composer><name" \-> ""), (copy Name);
                  ("</\><years birth=" \-> ""), (copy Birth);
                  (" death=" \-> ""), (copy Death);
                  ("/><nationality" . Nationality . "</</>" \-> "")]

let comps : lens = comp* . copy ""

let db : lens = ("<composers" \-> "") . comps . ("</>" \-> "")
Our syntax can write this same transformation as follows. Note the addition of the `grammar` and `end` keywords to designate the use of grammars:

```ocaml
define composer =
      name: String
      yearsBirth: Integer
      yearsDeath: Integer
      nationality: String

let comp : lens =
  grammar
  comp ::= n:Name , " b:Birth "~" d:Death ", " Nationality
    <>
    "<composer><name"> n:Name "</>"
    "<years birth=" b:Birth " death=" d:Death "/>"
    "<nationality>" Nationality "</></>"
end

let comps : lens =
  grammar
  comps ::= c:comp "\n" cs:comps <> c:comp "\n" cs:comps
  | "" <> ""
end

let db : lens =
  grammar
  db ::= cs:comps <> "<composers>" cs:comps "</>"
end
```

While this particular example requires more lines of code as a grammar, it is easy to see that the transformation is a lot easier to read and comprehend.

It is apparent that, while this program could be useful, it could be challenging for a new user to simply start writing Boomerang code, which is where our additions to the compiler could remove a significant obstacle. For the exact syntax specification of the grammars, see the first couple of chapters of the Boomerang Programmer’s Manual included as an appendix. Section 2.4 describes grammars in great detail.

As the complexity of the transformation increases, so does the Boomerang code. For instance, dealing with data records, where the order of the fields is different between the input and output, is syntactically difficult to implement in Boomerang, as evidenced by the examples in the appendix. This is an important reason why the addition of grammars is a worthy extension.

This system originally evolved from a standalone XSugar to Boomerang compiler. Using the OCaml language, we first defined intermediate representations of XSugar productions as well as Boomerang lenses. The XSugar production type definition was based largely on the grammar briefly described in the Quotient Lenses paper. We extended this into a full compiler, in which we parsed an XSugar grammar as input, and then passed over each production, collecting information relevant to the structure and ordering of data contained in both sides of the XML to ASCII transformation, and storing them in records. A second pass over the records allows us to complete the compilation to a Boomerang lens. We’ve tested this compiler with an example XSugar production and sample XML inputs, observing the expected ASCII output when the `get` function is invoked on the lens.
This compiler was relevant because it enabled us to gain experience writing a small compiler, and it served as a guide as to how certain transformation nuances are represented in Boomerang. This served us well during the next phase of our project.

After this was completed, the focus was on adding this functionality to Boomerang. The first step was to design the syntax we wanted to support. This was done collectively by Benjamin C. Pierce, Nate Foster, and ourselves. The syntax needed to be simple, lightweight, and easily readable while still being seamless with current Boomerang constructs. Once the syntax was designed, we were able to proceed with our development.

Integrating this compiler with Boomerang required a few things, the first of which was the modification of Boomerang’s parser. We extended the abstract syntax tree with our grammar types, replacing some of our previous constructs with already created Boomerang types and conventions. After the abstract syntax tree was complete, we began modification of the type checker. Since our grammar extends the range of possible inputs, the typechecker needed to be modified to prevent further malformed input. After this we focused on elaborating the type checker to act as an interpreter. Given the rigorous amount of type checking as well as the changes to our data structures resulting from integration, our actual compiler code changed quite a bit from our initial XSugar compiler, although the high level logic was essentially the same. We needed to have functionality for right-recursive grammars, calculate dependencies among productions, deal with reordering of data, and ultimately produce a lens.

The final stage of our project consisted of rigorous documentation. Since Boomerang is a language in production, our addition of grammar support needed to be noted. See the Appendix for the documentation. This gives the high level technical detailing of how grammars relate to lenses as well as some further examples.

Throughout the project there were many technical challenges. In the beginning the principle technical challenges were centered around familiarizing ourselves with bidirectional programming constructs. For instance, when we were creating our first compiler, we came across some intricacies of the Boomerang language, regular expressions in particular, that were initially a bit confusing, as we changed from one regular expressions library to another, adapting to the changes in available features. Also, our initial compiler attempted to produce an “intermediate lens” after the first pass over the XSugar production, which presented some difficulties, as critical information to producing a lens could not be obtained during this first pass. We adapted to using records as an intermediate representation, which has proved to be a much more suitable data type for the job.

There were also a tremendous amount of technical challenges during the Boomerang integration. This is where most issues arose. Given that Boomerang contains a large formed code base, where most constructs are foreign to us, there was a tremendous amount of learning that went on. Debugging resulted in a lot of trial and error when determining which functions we needed and what types were required for various functions. A large amount of our code needed to be re-written as a result of this.
Additionally, the Boomerang system is built on a constantly evolving codebase. Since the beginning of the project in the fall, many changes have been introduced to the language. Staying on top of these changes caused some headaches at times because it made debugging our code slightly more difficult, and sometimes difficult to determine if errors were the result of our code or changes made by others.

Another problem we ran into was that with our initial syntax design the parser had a lot of shift/reduce conflicts, resulting from ambiguities in the grammar. The parser was not able to determine where one production was ending, and the other was beginning. To solve this problem we added support for the keyword *and* to separate them.

### 3. Conclusion

During the course of this project, we became familiar with bidirectional programming, the Boomerang language, XSugar, functional programming constructs, as well as ocamllex and ocamlyacc. We accomplished many tasks, building upon one another before achieving our final goal. We began by learning about lenses and writing some sample Boomerang code. We then studied XSugar and wrote our initial compiler all the while becoming more and more familiar with OCaml and functional programming.

At the beginning of the project we didn’t anticipate to run into as much theory as we did. For instance, compiling grammars and computing dependencies among productions proves to be very tricky. It is one thing to understand it on the chalk board, but then to go ahead and implement it in a language that you are still learning is rather challenging. Integrating with Boomerang also proved to be trickier than we anticipated because of the small amount of code that could ultimately be reused. Boomerang integration added an entirely new dimension to the learning process as well, when this foreign language was dropped in our laps.

In the end, our technical skills greatly improved. Not only did we become proficient at functional programming and use a lot of theory we never thought we would, but we gained skills with managing a shared, evolving codebase. The amount of bugs we encountered also tested our patience to extreme levels.

### 4. Appendix

Attached are the first two Chapters of the Boomerang Programmer’s Manual. They serve as an introduction to Boomerang, programming with lenses, and provide numerous examples, all in full technical detail. Section 2.4 describes our contributions in detail, and should be treated as a major component to this report.

The complete manual, along with more information about the current status of Boomerang, can be found at [http://www.seas.upenn.edu/~harmony](http://www.seas.upenn.edu/~harmony).
5. References


The authors of this paper, researchers at the University of Pennsylvania, Ecole Polytechnique, and INRIA Rhone-Alpes, describe the Boomerang bidirectional programming language. Of particular relevance is their description of lenses as well as Boomerang’s get, put, and create functionality. This paper is relevant to our project because it forms the basis for our understanding of lenses and bidirectional languages. It also describe some extensions and future work that will be undertaken with the language.

Brabrand, Moller, and Schwartzbach (2005). Dual Syntax for XML Languages. *Proc 10th International Workshop on Database Programming Languages, DBPL ’05*

The frequency of this paper’s citation indicates that it is high-quality research. From researchers at the University of Aarhus in Denmark, this work provided a lot of insight into the XSugar programming language. It describes why there is a need for a language that transforms between ascii and XML. This paper served as the basis for much of our discussion about ascii and XML transformations.


The authors, researchers at the University of Pennsylvania and Ecole Polytechnique and INRIA, describe the importance of quotient lenses. Insight was gained regarding a lot of the details about XSugar grammars and transforming between two different types of lenses. Of particular use was the later sections of this paper. This wound up being our most valuable reference.


This paper, describe by researchers at Universite Paris 7 and Universite Paris-Sud 11, discuss the issues with XML bidirectionalism and proposes a small language of combinators to fix these issues. More specifically, the problems with document types after transformations by iterators is discussed. This resource, while not always directly applicable to our efforts, has served to keep us aware of typing issues during transformations.
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...

```xml
<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
$C$ of “concrete” strings and a set $A$ of “abstract” ones has three components:

\[
\begin{align*}
  l.\text{get} & \in C \rightarrow A \\
  l.\text{put} & \in A \rightarrow C \rightarrow C \\
  l.\text{create} & \in A \rightarrow C 
\end{align*}
\]

$\text{get}$ is the forward transformation and is a total function from $C$ to $A$. The backwards transformation comes in two flavors. The first, $\text{put}$, takes two arguments, a modified $A$ and an old $C$, and produces an updated $C$. The second, $\text{create}$, 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 $\text{get}$ 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 $a \in A$:

\[
\begin{align*}
  l.\text{put} \ (l.\text{get} \ c) \ c & = c & \quad \text{(GETPUT)} \\
  l.\text{get} \ (l.\text{put} \ a \ c) & = a & \quad \text{(PUTGET)} \\
  l.\text{get} \ (l.\text{create} \ a) & = a & \quad \text{(CREATEGET)}
\end{align*}
\]

The first law requires that if $\text{put}$ is invoked with an abstract string that is identical to the string obtained by applying $\text{get}$ 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 $\text{put}$ and $\text{create}$ 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 primitives for copying (copy) and deleting (del) strings, and for concatenating lenses (\( . \)).

```plaintext
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
>>
... 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. 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 (??) and on lenses in general (??). 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 "k" 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
Let’s define the lens that was used to generate this text.

```ocaml
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:

```ocaml
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.

```ocaml
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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
let a : 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:

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

```haskell
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 ?. To verify it on this example, try out this unit test:

```haskell
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 #{}[2;1;0]  
    #{}[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:LAST ", " fn:FIRST  
    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:LAST "", " fn:FIRST
               | fn:FIRST " " ln:LAST ";" ns:swap_many
               <- ln:LAST ";" fn:FIRST ";" ns:swap_many
  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:swap
               | n:swap ";" ns:swap_many
               <- n:swap ";" ns:swap_many
  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:

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

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

and verify correct behavior:

```haskell
let two_names : string =
  "John Smith, 24, M
  Jane Doe, 23, F"
```

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

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

```haskell
  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:FIRST " and " fn2:FIRST " " ln1:LAST
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, comps) : lens * lens =
  grammar
  comp ::= nm:ALPHA "," YEARS ""," cntry:ALPHA
    <-> nm:ALPHA ""," cntry:ALPHA
  and comps ::= c:comp <-> c:comp
    | c:comp newline cs:comps <-> c:comp newline cs:comps
end

and verify it with the same unit tests as earlier:

test comps.get c = a

## 2.4.2 Extending Grammars to Support Mutual Recursion

One might consider extending the Boomerang system to support grammars in which two productions are mutually recursive, as in the following (unsupported) example:

```
(*

let (pos,neg) : lens * lens =
  grammar
  pos ::= "positive" <-> "+
    | "positive" n:neg <-> "+" n:neg
  and neg ::= "negative" <-> "-
    | "negative" p:pos <-> "-" p:pos
end
```
While this would be a compact way to write a lens for terms that alternate, the same lenses can be expressed within the context of the current system as follows:

```haskell
let p : lens = del "positive" . ins "+" 
let n : lens = del "negative" . ins "-"

let cp_p : lens = copy " " . p 
let cp_n : lens = copy " " . n

let pos : lens = p . (cp_n . cp_p)* . cp_n? 
let neg : lens = n . (cp_p . cp_n)* . cp_p?
```

and indeed this is easily verifiable from the above tests:

```haskell
test pos.get "positive" = "+

test pos.get "positive negative positive negative" = "+ - + -

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

One can use Gaussian elimination to create a set of lenses for any such example, and depending on how difficult this proves to be to the programmer, mutually recursive productions may be supported in the future.