Elision User Guide

Version Build 14 March 2013

nominatim

Linguistically, an *elision* (i'liZHən) is the omission of sounds from an utterance to create a result that is (typically) easier to pronounce. Contractions are examples of elisions. Elision can also refer to omissions in books and films. Finally, it may mean the process of joining or merging together abstract ideas.

Contents

Elision is distributed under the following modified BSD “two clause” license.

```
      _ _     _
  ___| (_)___(_) ___  _ __
 / _ \ | / __| |/ _ \| '_ \
|  __/ | \__ \ | (_) | | | |
 \___|_|_|___/_|\___/|_| |_|
The Elision Term Rewriter
Copyright (c) 2012 by UT-Battelle, LLC.
All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
Collection of administrative costs for redistribution of the source code or
binary form is allowed. However, collection of a royalty or other fee in excess
of good faith amount for cost recovery for such redistribution is prohibited.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER, THE DOE, OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 1

Installation

This chapter explains how to obtain the Elision rewriter. Information about the rewriter can be obtained by visiting `http://elision.github.com/`. This page contains a frequently updated version, along with documentation links and a link to Elision's source.

Elision contains Parboiled, a parser expression grammar (PEG) library. See `https://github.com/sirthias/parboiled/wiki` for details, including the license agreement. The Elision REPL uses JLine, a Java library for handling console input. See `https://github.com/jline/jline2` for details, including the license agreement. Note that Elision actually uses the modified version of JLine built for the Scala REPL and distributed with Scala. See `http://www.scala-lang.org/`.

1.1. Running

If you only want to run the Elision REPL, you have two choices.

- Make sure the Java 7 runtime is installed and download the large `elision-all.jar`.[1] Run Elision with the command `java -jar elision-all.jar`. In some envrionments you can just double-click the jar file.
- Make sure both Java 7 and the Scala 2.9.2 (or later) distributions are installed. Download the (much smaller) `elision.jar` and run Elision with the command `scala elision.jar`.

1.2. Building

Elision requires Scala 2.9.2 or later to build. Be careful with later versions of Scala as the language is still changing quite a bit. The Java 7 (or later) SDK and Apache Ant 1.8 (or later) are also prerequisites.

(1) Install the Java 7 (or later) SDK. Visit `http://java.oracle.com` to download the correct version for your platform.
(2) Install Scala 2.9.2 (or later). Visit `http://www.scala-lang.org` to download the Scala distribution.
(3) Set the environment variable `SCALA_HOME` to point to the root folder of your Scala installation (the folder that contains the `bin` and `lib` folders).
   (a) On Windows platforms you should edit the environment variable settings under System, and add `%SCALA_HOME%\bin` to the `Path` environment variable. Open System Properties and click Advanced system settings. Then on the Advanced tab click Environment Variables... Under System Variables click New...

Variable name	Variable value
`SCALA_HOME`	`C:\scala`

   You should replace `C:\scala` with the directory where you installed Scala. Next edit the value for `Path` *carefully* to add `;%SCALA_HOME%\bin` on the end.

[1] Actually the filename will contain the build information, and will probably resemble `elision-all-20120601.jar`.

(b) On MacOS X platforms[2] you can edit the environment plist *or* follow the directions for UNIX below. You can do this from the Terminal prompt with `open ~/.MacOSX/environment.plist`. This will open the environment settings in the Property List Editor. Add a new Item at the top level.

Key	Type	Value
SCALA_HOME	String	/usr/local/scala

You should replace `/usr/local/scala` with the directory where you installed Scala. Next edit the value for `PATH` *carefully* to add `:$SCALA_HOME/bin` on the end.

(c) Many modern Linux systems employ a modular profile system in which you can create a `/etc/profile.d/scala.sh` file with the following content:

```
# Add Scala
export SCALA_HOME=/usr/local/scala
PATH=${PATH}:$SCALA_HOME/bin
```

You should replace `/usr/local/scala` with the directory where you installed Scala.

(d) As a last resort you can add code to `/etc/profile`, if it exists. The code to add is precisely the same as the code shown above for the modular profile. If you are unable to edit the global `/etc/profile`, you may instead edit your own `~/.bash_profile` and add those lines. This last case appears to work well on modern Mac OS X systems.

(4) Install Apache Ant 1.8 (or later). Visit `http://ant.apache.org` to download the Ant distribution.

(5) (Optional) If you want to use Eclipse, you should install Eclipse Juno (or later).

(6) To build the Elision jar file, `cd` to the root folder of the Elision distribution and run the command `ant`. This will build the jar file.

(7) (Optional) If you want to build the API documentation, then run the command `ant docs`. To build both the jar files and documentation, use `ant all`. To see what other targets are available, use `ant -p`.

1.3. REPL

Elision comes with a read, evaluate, print loop interpreter (a REPL). You can start it in any of the following ways.

- Use the `elision.sh` (for UNIX / Linux / Mac OS X) and `elision.bat` (for Windows) scripts found in the root folder of the Elision distribution. This starts the REPL using the compiled class files in the `bin` folder, and is the best way to run it if you are using a continuous build system such as Eclipse. If you try to run the REPL this way while Eclipse is compiling, you will get missing class file errors. Just wait for Eclipse to finish compiling.
- To run the REPL from the jar file, execute the command `scala elision.jar`. You must have built the jar file to use this method, and must update it each time you make changes.
- To start the REPL from inside a program, invoke the `ornl.elision.repl.ReplMain.runRepl()` method. For instance, there are `run.sh` and `run.bat` scripts in the root of the Elision distribution that will start a Scala REPL with all Elision classes (if Elision has been built) in the classpath. You can then invoke `ornl.elision.repl.ReplMain.runRepl()` directly from the Scala prompt.

To exit from the REPL use `:quit`. To get help use `help()`. The REPL is the subject of chapter 2.

[2] It seems that, with Mountain Lion, OS X no longer observes the settings in the `environment.plist`. In this case you may need to follow step d below instead of using the instructions given here.

1.4. Scala with Emacs, Notepad-Plus, and Others

Most editors now come with a Scala mode installed. In case yours does not, you should look in the Scala distribution under `misc/scala-tool-support`. We will provide some instructions for Emacs here.

The Emacs mode works for Emacs 21.1or later, and does not work on XEmacs. Things change; consult the `README` file in the folder for current details. It is suggested you leave these files in place so they get magically updated when the next version of Scala is installed.

In your Emacs startup file (typically `~/.emacs` on Linux or UNIX) add the following two lines.

```
(add-to-list 'load-path
  "/usr/local/scala/misc/scala-tool-support/emacs")
(require 'scala-mode-auto)
```

Adjust the second line as necessary to point to your Scala installation. Now restart Emacs, and you should get the Scala mode when you open a file ending with `.scala`. You can compile the `.el` files to `.elc` files using the provided `Makefile`, but be sure to run `make` each time you update.

1.5. Scala with Eclipse

The root folder of the Elision distribution contains an Eclipse project. Before you attempt to import it into Eclipse you should do the following.

- Install the **Scala IDE**. At present the version of the Scala IDE for Scala 2.9.2 can be found at the update site http://download.scala-ide.org/releases-29/stable/site. Visit `http://scala-ide.org` for the latest version, documentation, etc.

That is all that is required. You may optionally install the following helpful plugins.

- **ShellEd** is an excellent editor for working with shell scripts, but at present it only works with Eclipse Indigo. You can find out more at `http://sourceforge.net/apps/trac/shelled`. To install ShellEd you need to install components from the Linux Tools. The update site is `http://download.eclipse.org/technology/linuxtools/update`. Add this update site first, but do not install anything. The update site for ShellEd is `https://downloads.sourceforge.net/project/shelled/shelled/ShellEd%202.0.1/update`. After adding the update site for the Linux Tools, add this update site and then install ShellEd; it will resolve the dependencies and get the components from the Linux Tools that are actually required.
- **Copyright Wizard** can be used to automatically and quickly update the copyrights on all files and to make sure all new files have the correct copyright information. The distribution includes a configuration file for Copyright Wizard that will be automatically discovered if you install Copyright Wizard. Find out more at `http://www.wdev91.com/?p=cpw`. The update site is `http://www.wdev91.com/update`.

Once you have installed the Scala IDE plugin, do the following.

(1) Unpack the Elision distribution into an Eclipse workspace, or create a `workspace` folder and place the `Elision` folder in it.
(2) Start Eclipse. You will probably be prompted to run the Scala Setup Diagnostics. Click Yes to run them. Make sure you open Eclipse in the correct workspace containing the Elision distribution.
(3) Go to the workbench. Right-click in the package explorer and choose New and then Project. In the New Project wizard expand Scala Wizards and select Scala Project. Click Next.
(4) In the New Scala Project dialog enter Elision as the project name. If you have done everything correctly you should see a note at the bottom of the dialog telling you the wizard is about to automatically configure the project based on the existing source. This is what you want, so click Finish.

(5) You may be asked if you want to switch to the Scala perspective. You do, so click Yes.

At this point Elision should automatically build, and you are ready to start working with the code or running the REPL using `repl.sh` or `repl.bat`.

1.6. Scala Code

Elision is written in Scala, and provides a Scala API. Discussions of the Scala API are highlighted in blue.

Scala. Elision's classes live in packages under `ornl.elision`. Throughout this document packages, classes, objects, traits, etc., will be discussed in reference to this top-level package. Thus `core.BasicAtom` refers to the fully-qualified class name (FQCN) `ornl.elision.core.BasicAtom`.

CHAPTER 2

Interacting with the REPL

The REPL is Elision's read, evaluate, print loop. It provides a way to interact directly with the rewriter.

2.1. Starting the REPL

You can start the REPL in any of the following ways.

- Use the `elision.sh` (for UNIX / Linux / Mac OS X) and `elision.bat` (for Windows) scripts found in the root folder of the Elision distribution. This starts the REPL using the compiled class files in the `bin` folder, and is the best way to run it if you are using a continuous build system such as Eclipse. If you try to run the REPL this way while Eclipse is compiling, you will get missing class file errors. Just wait for Eclipse to finish compiling.
- To run the REPL from the jar file, execute the command `scala elision.jar`. You must have built the jar file to use this method, and must update it each time you make changes.
- To start the REPL from inside a program, invoke the `ornl.elision.repl.ReplMain.runRepl()` method. For instance, there are `run.sh` and `run.bat` scripts in the root of the Elision distribution that will start a Scala REPL with all Elision classes (if Elision has been built) in the classpath. You can then invoke `ornl.elision.repl.ReplMain.runRepl()` directly from the Scala prompt.

All of these methods will display the Elision banner and present you with the REPL prompt `e>`. You can enter text to be evaluated as an Elision atom (described in later sections), interact with the history (using the up and down arrow keys), or edit the current line. Pressing Enter submits the line to the Elision parser for evaluation.

Scala. If you want to embed Elision in another system, then you may want to create your own "executor" to parse and execute operations. The Elision system expects to be able to find an instance of `core.Executor`. The purpose of `Executor` is the following.

- It maintains the *console* that gets output and provides input. The console must be an instance of `ornl.elision.util.Console`.
- It manages a *context* that maintains an operator library, a rule library, and a set of global bindings. The context must be an instance of `core.Context`.
- It can *parse* text in some form and return Elision atoms. The input text is a `String`, and the result is given as an instance of `Executor.ParseResult`.

A rather sophisticated version of `Executor` is provided by the `parse.Processor` class. This class actually implements most of what is needed by the `Executor`, along with several additional traits, and is geared toward interactive operations on persistent data. Finally, the Elision REPL is implemented by `repl.ERepl`, which extends `Processor` to add support for history, known properties, and interaction at a prompt.

If you create your own `Executor` instance, you must make sure to install it as the "known executor" prior to performing any parsing work. Do this with a line of the

form `ornl.elision.core.knownExecutor = myExecutor`, where `myExecutor` is your customized `Executor` instance.

The most useful commands to know immediately are `help()` (to get help) and `:quit` (to quit).

The REPL commands (with the exception of `:quit` and history recall) are actually Elision operators (see chapter 5). As such you must follow them with parentheses, otherwise the Elision REPL will interpret them as symbols. Try `history` and then try `history()`.[1]

The REPL reads zero or more atoms per line, evaluates them, and displays the output. Several atoms can be present on a single line; in that case the REPL displays each one.

Atoms can span multiple lines. If a parenthesis, brace, bracket, or verbatim block[2] is open at the end of the line, then the REPL will prompt for additional input with the prompt `>`. The newlines are converted to spaces, except with verbatim blocks.

```
e> { operator
 >   #name=set
 >   #params=%ACI($P: BOOLEAN, $Q: BOOLEAN)
 >   #type=BOOLEAN
 > }
$_repl0 = {:operator:SYMBOL { binds name -> set:SYMBOL
  params -> %ACI($P:BOOLEAN, $Q:BOOLEAN) type -> BOOLEAN }:}
e> ("Fred".
 > " Flintstone")
$_repl1 = "Fred␣Flintstone"
```

2.2. History

The Elision REPL saves a persistent history from session to session. This history can be viewed with the `history()` command.

```
e> history()
0: // New Session: Fri Jun 08 12:12:59 EDT 2012 Running: 0.1, build 201206081150
1: {operator #name=test #params=%($x:INTEGER) #type=INTEGER}
2: {rule add($x,neg($x)) -> 0}
3: add(21,17,$x)
4: history()
Persistent history is found in: /home/userguy/.elision-history.eli
```

[1] It is permissible to put spaces between the operator name and the opening parenthesis: `history ()`. This can lead to trouble. Suppose you want to have the symbol `fred` followed by a parenthesized expression: `fred (5).$x`. Here the parentheses are necessary to prevent the parser from interpreting the dot as a decimal point. Unfortunately this will not work the way you might expect. The system will parse this as `(fred(5)).$x`. To prevent this, put parentheses around `fred`: `(fred) (5).$x`.

[2] A verbatim block is text within triple double quotation marks: """..."""". No escapes are interpreted, and spaces and newlines are maintained. Leading spaces are removed. Thus the verbatim block:

```
"""This is the first line,
  and this is the second.
Escapes and quotation marks are preserved.
"See \n?" he said."""
```

Denotes the following string.

```
"This is the first line,\nand this is the second.\nEscapes and quotation marks
are preserved.\"See \\n?\" he said."
```

It is possible to recall any prior history element by entering the history index after an exclamation mark. Entries that span multiple lines are not properly recalled in this manner, however.

```
e> !3
add(21,17,$x)
$_repl14 = add($x, 38)
e> bind($x, 2)
Bound $x
e> !3
add(21,17,$x)
$_repl15 = 40
e> history()
0: // New Session: Fri Jun 08 12:12:59 EDT 2012 Running: 0.1, build 201206081150
1: {operator #name=test #params=%($x:INTEGER) #type=INTEGER}
2: {rule add($x,neg($x)) -> 0}
3: add(21,17,$x)
4: history()
5: add(21,17,$x)
6: bind($x, 2)
7: add(21,17,$x)
8: history()
Persistent history is found in: /home/userguy/.elision-history.eli
```

2.3. Properties

The operation of the REPL, and of Elision itself, is configurable using *properties*. To see all properties defined in the current system, use the command `listprops()`. This lists each property name, gives a short description, and gives the *current* value of the property in parentheses.

```
e> listprops()
applybinds ..... Rewrite each atom with the
                 context's␣bindings.␣␣(true)
autoop␣.................␣If␣the␣current␣result␣is␣an
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣operator,␣automatically␣declare␣it
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in␣the␣operator␣library.␣␣(true)
autorewrite␣.............␣Automatically␣apply␣rules␣in␣the
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣active␣rulesets␣to␣each␣atomas␣it
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣is␣evaluated.␣␣(true)
autorule␣...............␣If␣the␣current␣atom␣is␣a␣rewrite
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rule,␣automatically␣declare␣it␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣the␣rule␣library.␣␣(true)
...
```

To set the value of a property, use `setprop(name,value)`, where `name` is the property name (a string) and the value is an atom of the correct type.

2.4. Binding

We can tell the Elision system that we want to replace the variable `$x` with the value `7` wherever it appears. We say the variable `$x` is *bound* to `7`, and that the pair `$x` and `7` is a *binding*.[3] When we replace the variable `$x` with `7` in some other term, we say we are *rewriting* the term with the binding. The REPL keeps a set of bindings that are used to immediately rewrite any atom *after*

[3]This is similar to, but distinct from, a *map pair*. Map pairs are explained in section §7.3.

it is parsed and constructed. For instance, if the variable `$x` is bound to `7`, then entering `$x` at the prompt results in the value `7`. These bindings are stored in the REPL's *context*. Contexts are explained in **??**; for now you only need to know that the REPL creates and maintains a context for you.

The `bind(`*v*`,`*a*`)` command binds a variable *v* to an atom *a*. For instance, try `bind($x,7)` and `bind($y,\$x.%($x,$x))`. Then try `$x` and `$y` at the prompt. Then try `$y.$x`. Variables can be rebound at any time, and there is no need to "unbind" them first.

```
e> bind($x,7)
Bound $x
e> bind($y,\$x.%($x,$x))
Bound $y
e> $x
$_repl16 = 7
e> $y
$_repl17 = \$‘:1‘.%($‘:1‘, $‘:1‘)
e> $y.$x
$_repl18 = %(7, 7)
```

The `unbind(`*v*`)` command removes any binding for the variable *v*. Try `unbind($x)` and then repeat `$y.$x`. No error is generated if a variable is not bound.

```
e> unbind($x)
Unbound $x
e> $y.$x
$_repl19 = %($x, $x)
```

If the `setreplbinds` property is `true` (which is the default), then every time an atom is evaluated by the REPL, the result is displayed and then bound to a new variable whose name starts with `$_repl` followed by a number. These variables are displayed on the output. Try `$_repl1`. Unlike the history, which is persistent between invocations of the REPL, these variables are re-started at each session.

To see all the bindings being used by the shell, enter the command `showbinds()`.[4]

2.5. Atom Display

When you type an atom at the REPL prompt the system parses the atom and constructs it, which may involve evaluating operators. Otherwise the current bindings are applied to the atom, and the atom is then rewritten using any active rulesets. The result of this is then bound to the next `$_repl`*n* variable.

You can see the atom after it is parsed and constructed, but before bindings are applied and rewriting is performed, by setting the `showprior` property to `true`.[5]

```
e> inc("bootstrap/Math.eli")
e> bind($x,10)
Bound $x
e> setprop("showprior", true)
‘ NO SHOW ‘:SYMBOL
e> add($x,$x,0)
add($x, $x)
```

[4]Note that any bindings to variables whose name starts with an underscore are suppressed in the output.

[5]This action results in the odd output `‘ NO SHOW ‘:SYMBOL`. Elision expects every operator to yield an atom, but this makes no sense for operators executed only for their side effect. As such, these operators return a special atom typically hidden by the REPL. Enabling `showprior` causes ths atom, `‘ NO SHOW ‘:SYMBOL`, to be displayed.

```
$_repl20 = 20
e> setprop("showprior", false)
```

The atom as entered was `add($x,$x,0)`.[6] During construction the identity `0` is removed, leaving the atom `add($x,$x)`, which is displayed. Then the bindings are applied, yielding `add(10,10)`, which is rewritten to `20` and displayed as the final result.

Most REPL printing can be suppressed using the `quiet(level)` command. The `level` is one of the following values.

0	Show all output.
1	Suppress most output, but still show errors, warnings, and explicitly requested output.
2	Suppress warnings, but show errors and explicitly requested output.
3	Suppress warnings and errors, but still show explicitly requested output.
4	Suppress nearly everything.

Any quiet level except 0 changes the prompt to `q>`.

Scala. It is possible to display the Scala code to create the atom. This is enabled (and disabled) with the `showscala` property. The assumed context for the resulting code is that the content of the `core` package is imported.

```
e> setprop("showscala", true)
e> \$x:$T.$T
Scala: $_repl28 = Lambda(Variable(Variable(NamedRootType("ANY"), "T",
  BooleanLiteral(NamedRootType("BOOLEAN"), true), Set()), ":1",
  BooleanLiteral(NamedRootType("BOOLEAN"), true), Set()),
  Variable(NamedRootType("ANY"), "T", BooleanLiteral(NamedRootType("BOOLEAN"),
  true), Set()))
$_repl28 = \$‘:1‘:$T.$T
```

This can be useful to discover how to construct an atom using the Scala API.

If the above seems excessively complex, *don't panic*. There are several helpful parts of the API, including implicit conversions, to make this all much simpler.

2.6. Automatic Rewriting

Once an atom is entered and bindings are applied, it is rewritten with the active rulesets. (Applying rules is the subject of chapter 8.)

Rulesets are initially disabled when declared. They can be *enabled* using the `enable(RS)` command, where `RS` is the name of the ruleset. Likewise, they can be *disabled* with the `disable(RS)` command.

```
e> decl.{! foo($x,$y)}
e> decl.{rule foo(0,$x) -> $x #rulesets RENAME}
ERROR: The ruleset RENAME has not been declared.
e> declare(RENAME)
Declared ruleset RENAME:SYMBOL.
e> decl.{rule foo(0,$x) -> $x #rulesets RENAME}
Declared rule.
e> foo(0,14)
$_repl8 = foo(0,14)
```

[6] The `add` operator is defined in the `bootstrap/Math.eli` file that is not normally read into the system. The `inc("bootstrap/Math.eli")` instruction causes these math definitions to be available.

```
e> enable(RENAME)
e> foo(0,14)
$_repl9 = 14
```

A limit is placed on the number of times a rule can be applied to rewrite an atom. This limit can be set via the `setlimit(lim)` command, where `lim` is the new limit. A limit of zero disables automatic rewriting, and a negative limit is equivalent to no limit. Automatic rewriting is on by default, but can be disabled (or re-enabled) altogether with the `autorewrite` property. It is enabled by default.

You can ask the system which rules may be applied to an atom, using the `showrules(atom)` command, where `atom` is the atom to test.

```
e> showrules(foo(5,4))
{:rule:SYMBOL { binds `` -> %((foo(0,$x) -> $x)) rulesets -> %(RENAME:SYMBOL) }:}
```

The output looks quite different from the rule as it was originally entered. This is because rules are a *special form*, and Elision displays the two atom form for special forms. This is explained in chapter 6.

2.7. Understanding Matching

To support better understanding of the matching process, it is possible to enable (or disable) tracing the matching process with the `tracematch()` command. When enabled considerable diagnostic information is printed for every match that is attempted.

Several things trigger the matching system. One is constructing an operator application, where arguments are matched against the formal parameters. Another is matching a lambda argument to the parameter. Of course, the most obvious way is by explicitly trying to match two atoms.

```
e> inc("bootstrap/Math.eli")
e> tracematch()
Match tracing is ON.
e> \$x.add(1,$x).5
TRYING  (1412e2c9) in class ornl.elision.core.Variable:
  pattern: $x:INTEGER
  subject: 1
  with: { binds }
TRYING  (f021a3c0) in class ornl.elision.core.INTEGER$:
  pattern: INTEGER
  subject: INTEGER
  with: { binds }
SUCCESS (f021a3c0): { binds }
SUCCESS (1412e2c9): { binds x -> 1 }
...
TRYING  (1412e2ce) in class ornl.elision.core.Variable:
  pattern: $x:INTEGER
  subject: 6
  with: { binds }
TRYING  (f021a3c0) in class ornl.elision.core.INTEGER$:
  pattern: INTEGER
  subject: INTEGER
  with: { binds }
SUCCESS (f021a3c0): { binds }
```

```
SUCCESS (1412e2ce): { binds x -> 6 }
$_repl14 = 6
```

Reading this trace can be tricky. The lines that begin with `TRYING` and `SUCCESS` or `FAILURE` are linked using the hexadecimal number in parentheses. This number is actually a hash code for the match attempt and serves to link together the lines. In a large match attempts will often interleave.

Following a `TRYING` line the pattern, subject, and any bindings are printed. On `SUCCESS` we see the bindings that result in a successful match. On `FAILURE` we are told why the match failed, and the pattern and subject may be repeated.

In some cases there will be additional matching reports after the `$_repl` result is shown. These are due to *round-trip testing* by the system. This is testing that determines whether or not an atom's representation can be successfully re-parsed to yield the original atom, and this output can safely be ignored. Look for the line beginning with `$_repl` and ignore subsequent output. To disable round trip testing, set the property `roundtrip` to `false`.

```
e> {match 7}.$x
Sequence Matcher called:
    Patterns: $domain,$codomain
    Subjects: ANY,BINDING
    Bindings: { binds }
TRYING  (71407c54) in class ornl.elision.core.Variable:
  pattern: $domain
  subject: ANY
  with: { binds }
SUCCESS (71407c54): { binds domain -> ANY }
TRYING  (f6b11b01) in class ornl.elision.core.Variable:
  pattern: $codomain
  subject: BINDING
  with: { binds domain -> ANY }
SUCCESS (f6b11b01): { binds domain -> ANY codomain -> BINDING }
TRYING  (c390faa3) in class ornl.elision.core.IntegerLiteral:
  pattern: 7
  subject: $x
  with: { binds }
TRYING  (50a1946e) in class ornl.elision.core.INTEGER$:
  pattern: INTEGER
  subject: ANY
  with: { binds }
SUCCESS (50a1946e): { binds }
FAILURE (c390faa3): Literal pattern does not match subject.
  pattern: 7
  subject: $x
$_repl10 = NONE
```

For this simple example we see the inner attempt to match the types succeeds, and then the outer attempt to match the entire atoms fails because `7` is a literal pattern and it can only match another literal `7`.

2.8. Understanding Parsing

The parser can also be traced, and this results in an enormous quantity of output. To enable (or disable) parser tracing, use the `traceparse()` command.

```
e> traceparse()
Parser tracing is ON.
e> 7
Starting new parsing run
a sequence of atoms/ZeroOrMore/AtomSeqPush1Action1, matched, cursor at 1:1 after ""
..(1)../ZeroOrMore/ZeroOrMore/Sequence/an atom/LApply/a simple atom/whitespace or
comments/FirstOf/OneOrMore/[ \n\r\t\f], failed, cursor at 1:1 after ""
..(9)../OneOrMore, failed, cursor at 1:1 after ""
..(8)../FirstOf/Sequence/"/*", failed, cursor at 1:1 after ""
..(9)../Sequence, failed, cursor at 1:1 after ""
..(8)../FirstOf/Sequence/"//", failed, cursor at 1:1 after ""
..(9)../Sequence, failed, cursor at 1:1 after ""
..(8)../FirstOf, failed, cursor at 1:1 after ""
..(7)../whitespace or comments, matched, cursor at 1:1 after ""
..(6)../a simple atom/FirstOf/Sequence/'(', failed, cursor at 1:1 after ""
..(8)../Sequence, failed, cursor at 1:1 after ""
..(7)../FirstOf/a ruleset declaration/'{', failed, cursor at 1:1 after ""
..(8)../a ruleset declaration, failed, cursor at 1:1 after ""
..(7)../FirstOf/a rewrite rule/'{', failed, cursor at 1:1 after ""
..(8)../a rewrite rule, failed, cursor at 1:1 after ""
...
..(3)../Sequence, failed, cursor at 1:2 after "7"
..(2)../FirstOf/Sequence/"//", failed, cursor at 1:2 after "7"
..(3)../Sequence, failed, cursor at 1:2 after "7"
..(2)../FirstOf, failed, cursor at 1:2 after "7"
..(1)../whitespace or comments, matched, cursor at 1:2 after "7"
a sequence of atoms/EOI, matched, cursor at 1:3 after "7"
a sequence of atoms, matched, cursor at 1:3 after "7"
$_repl14 = 7
```

Many, many lines are omitted from the above. The number shows the depth of the parse which, for this example, reaches 16. Note that, as with match tracing, the round trip testing results in additional output that can be ignored.

2.9. Reading Files

If you want to load several operator definitions and rules, you can put these in a file and then tell Elision to read the file. You do this via the `read(filename)` operator, where `filename` is the name of the file to load. Elision reads the contents of the file, processing bindings, operator definitions, rules, etc. Most output from this process is suppressed by first putting the REPL is quiet mode and suspending creating `$_repl` bindings.

If the filename is absolute (that is, it fully specifies the location of the file), then the file is simply read. Otherwise Elision needs to *find* the file before it can read it. It does this by checking a search path. The path is specified by the `path` property. This is a string containing a list of folders to search. On Windows the folders are separated by semicolons (`;`), while on UNIX (including Mac OS X) and Linux they are separated by colons (`:`).

2.10. Bootstrapping

When the Elision REPL starts, it first reads its bootstrap files that declare all the "built in" operators and rules.[7] It does this by locating and reading the file `Boot.eli`. This file provides *minimal* capabilities. A few critical operators are declared: `decl`, `inc`, `print`, and `println`. After this, the `inc` operator is used to include additional files, and the `decl` operator is used to declare operators and rules. As such, `Boot.eli` is a bit unusual.

The `Boot.eli` file includes (via `inc`) a few other files. `Type.eli` provides the type extraction operator, `Core.eli` provides several common operators, and `Context.eli` provides operators for the current context and controls for rule rewriting.

Scala. The `repl.ERepl` class is responsible for loading the `Boot.eli` file and kicking off the bootstrap process. If you replace this with your own executor instance (as described in the Scala block of section §2.1) then you are responsible for all bootstrapping. The most common symptom of improper bootstrapping, or of failing to correctly set `knownExecutor` as the first operation, is errors regarding `decl`.

[7] This includes declaring the operator `decl` that is used to declare other operators and rules. How this is done is explained in appendix D.

CHAPTER 3

Basic Atoms

The objects manipulated by Elision are called *atoms*, and they exist in several forms.

- *Literals* that denote a simple, fixed value: `5`, `"James"`, `1.9e3`, `test`.
- *Variables* denote an unspecified atom, perhaps with type information: `$x`, `$y:INTEGER`, `$$z`.
- *Lambdas* denote simple unnamed functions of one parameter: `\$x.add(1,$$x)`.
- *Applies* denote an ordered pair consisting of a left-hand element often called the *operator* and a right-hand element often called the *argument*: `foo(4,5)`, `add.%(1,2)`.
- *Operators* denote a mapping: `decl:OPREF`.
- *Rewriters* transform an atom into a potentially new form, reporting whether this transformation succeeded: `{rule foo(0,$x) -> $x}`.

Atoms can be combined to make composite atoms[1] in a variety of ways. These are discussed in the following sections and chapters.

3.1. Types

Every atom has an associated *type*, which is itself just another atom. Since every atom has a type, and every type is an atom, we have a potential infinite regress. To avoid this the special *type universe* atom, denoted `^TYPE`, is its own type.[2] The fact that atom a has type T is denoted by joining the atom and its type with a colon: $a : T$. The type colon is right associative, so $a : B : C$ denotes $a : (B : C)$. In fact, this is the only way the colon can work. Since an atom has exactly one type, $(a : B) : C$ is effectively meaningless, and generates an error in Elision. The term *root type* refers to any type whose own type is `^TYPE`. For instance, `STRING` is a root type that denotes `STRING:^TYPE`. If the symbol `STRING` is desired, it must be written `STRING:SYMBOL`.

Scala. Every atom is an instance of a class extending `core.BasicAtom`. All atoms are immutable after construction. The type of an atom must be specified at construction time, and can be accessed later via the public field `theType`. The type universe can be accessed in two ways: `core.TypeUniverse` is a singleton implementing the type universe, and it can be accessed via the alias `` `^TYPE` `` provided by the `core` package. Back ticks must be used since `^TYPE` is not a valid Scala symbol.

3.2. Literals

Elision directly supports several kinds of literals. These are indivisible, simple-valued atoms.

3.2.1. Symbols. Symbols start with an underscore or letter, followed by any number of letters, digits, and underscores. The following are legal symbol literals.

```
A          x          _21          Fred21
```

[1] No, not molecules. This is not chemistry.

[2] As has been pointed out, "it's turtles all the way down." `^TYPE` is the turtle that stands on its own head.

TABLE 1. Escape Codes

\f	Form Feed Character
\n	Newline
\r	Carriage Return
\t	Tab Character
\‘	Back tick
\”	Double Quotation Mark
\\	Backslash

Other characters may occur in symbols, but the entire symbol must be enclosed in back ticks that are not themselves part of the symbol. Several escape sequences are interpreted; these are listed in table 1. Using the back tick notation the following are also legal symbol literals.

```
‘1‘          ‘$_4‘          ‘\‘Fred\“          “
```

Scala. Symbols are created via the `core.Literal(sym)` method, where `sym` is a Scala symbol.

Take care! Scala symbols are automatically converted to Elision *variables*, not symbol literals, when used in a context expecting an Elision atom.

3.2.2. Strings. A string is a sequence of characters enclosed in double quotation marks that are not themselves part of the symbol. As with back-tick-quoted symbols, escape codes are interpreted in the string. The escape codes are listed in table 1. The following are legal string literals.

```
""      "Fred"      "‘21,‘ she said."      "\"Yes,\" he replied."
```

Elision also provides for verbatim blocks, which evaluate to strings. A verbatim block is indicated by any arbitrary text enclosed within triple quotation mark pairs. All characters are preserved as-is, and the result of evaluation is a string literal.

```
e> """
 > This is a multiline string literal.
 > Note that \ is left untouched."""
$_repl3 = "\nThis␣is␣a␣multiline␣string␣literal.\nNote␣that␣\\␣is␣left␣untouched."
```

Finally, strings can be concatenated with the applicative dot. Thus `"DJ "."Kousuke"` is interpreted as `"DJ Kousuke"`. This enables defining strings with variables, as with `{binds last->"Kousuke"}.("DJ ".$last)`, which also yields `"DJ Kousuke"`.

Scala. Strings are created via the `core.Literal(str)` method, where `str` is a Scala string.

Scala strings are automatically converted to Elision string literals as necessary. Thus it is possible to pass `"$"` to a method expecting an Elision atom.

```
scala> import ornl.elision.core.BasicAtom
scala> "I’m␣a␣string":BasicAtom
res1: ornl.elision.core.BasicAtom = StringLiteral(STRING,"I’m␣a␣string")
```

3.2.3. Numbers. Numbers may be expressed in binary, octal, decimal, or hexadecimal (the *radix*). Binary numbers are indicated by the prefix `0b`, octal numbers are indicated by a leading `0`, decimal numbers are indicated by a leading non-zero digit, and hexadecimal numbers are indicated by the prefix `0x`. The prefixes are *not* case-sensitive.

Floating point numbers are also supported in binary, octal, decimal, and hexadecimal, and are indicated either by a decimal point, an exponent, or both. The exponent is typically indicated by an `e`, must be an integer, and may be in a different radix. Because `e` is a valid hexadecimal digit, `p` may also be used to indicate an exponent (and *must* be for hexadecimal numbers). Numbers are *not* case sensitive. If a floating point number consists of significand s, exponent e, and radix r, then the number denoted is given by the following equation.

$$s \times r^e$$

The following are legal number literals.

`0`	`-1`	`05743645134635`	`-0x232`
`2.5`	`0b10.1`	`0x2.8`	`02.4`
`21e-6`	`0b10101e-0b110`	`0x15p-0x6`	`025e-06`

Keep in mind that the radix of the significand and the exponent must both be specified.

The number undergoes a conversion to place it in a consistent radix.[3] This conversion is performed as follows. Let the number be $i.f\texttt{e}e$, where i is the integer portion of the significand, f is the fractional portion of the significand, and e is the exponent. We note that $i.f$ is in one radix, which we denote r, and the exponent may be in a different radix. If the length of f is $|f|$, then the entire significand is multiplied by $r^{|f|}$ to convert it to an integer. The exponent is then converted to radix r, yielding e', and $|f|$ is subtracted from it.

$$n = i.f\texttt{e}e = (i.f)r^{|f|}\texttt{e}(e' - |f|)$$

Scala. Integers are created via the `core.Literal(i)` method, where `i` is a Scala `BigInt`. Floating point numbers can be created via the `core.Literal(FLOAT,sig,exp,rad)` method, where `sig` is the significand (a Scala `BigInt`), `exp` is the exponent (a Scala `Int`), and `rad` is the radix (one of 2, 8, 10, or 16).

Scala `Int` and `BigInt` are automatically converted to Elision integer literals as necessary. Thus it is possible to pass `17` to a method expecting an Elision atom.

If you have a floating point literal, you can obtain its value as a Scala float with `toFloat` and as a Scala double with `toDouble`. You can also convert it to radix two using `toBinary`. Because this changes the base of a floating point number you may lose precision.

At present Elision does not support the positive or negative infinity or the signaling or quiet NaN.

3.2.4. Booleans. Boolean literals come in two forms: `true` and `false`. Both have the type `BOOLEAN`. If you want an Elision symbol named `true` or `false`, you must override the type, and write `true:SYMBOL`, for instance.

Scala. The Boolean literals can be obtained as `Literal.TRUE` and `Literal.FALSE`. Scala Booleans are automatically converted to Elision Boolean literals as necessary.

3.3. Special Symbols

Certain symbols are interpreted differently from others. Three have already been discussed: `^TYPE` denoting the type universe, `true`, and `false`. Others denote named root types. A *root atom* is an atom whose type is `^TYPE`. table 2 lists the named root atoms declared by the current system.[4]

[3] This conversion takes place during parsing.

[4] It is possible to create new named root atoms. The following declares an operator that does just this. Give it a symbol, and that symbol becomes a named root type. For details about how this works, see appendix D.

TABLE 2. Special Root Type Symbols

Elision Symbol	Interpretation
ANY	The wildcard. The special atom denoting anything.
BINDING	The root type for all bindings.
BOOLEAN	The root type for all Booleans.
FLOAT	The root type for all floating point numbers.
INTEGER	The root type for all integer numbers.
NONE	The special atom denoting nothing.
OPREF	A root type for indicating that a symbol denotes an operator.
RSREF	A root type for indicating that a symbol denotes a ruleset.
STRATEGY	The root type for all strategies and rewrite rules.
STRING	The root type for all strings.
SYMBOL	The root type for all symbols.

These are typically used as types, so they are often called *root types*. Two of these – OPREF and RSREF – exist to cause the system to "look up" a name in the current context.

The special atom ANY matches any atom, and any atom matches it, so it can be treated as a wildcard during matching, and is also the assumed type of otherwise untyped atoms. ANY is unusual; it violates the substitution principle (see chapter 7). ANY matches NONE, but NONE matches only itself.

Untyped symbols not in the above list have the assumed type SYMBOL. If you want an ordinary symbol counterpart to any of the above, you must explicitly ask for it. For example, if you want the ordinary symbol STRATEGY, you must use STRATEGY:SYMBOL.

Scala. All the symbols listed in table 2 are also Scala objects that extend core.NamedRootType. They can be used directly, so SYMBOL is a Scala value denoting the Elision SYMBOL type.

3.4. Variables

Variables are indicated by a leading dollar sign followed by the variable name, which is parsed as a symbol. The following are legal variables.

```
$x          $_21          $‘1‘          $‘$_4‘
```

Variables are looked up by name, but matched by *both* name and type. This means that you should never use the same variable name with different types in a single atom, as it will break

```
decl.{! declare_rt($name: SYMBOL)
#description="Declare a new named root atom."
#detail="Declare $name to be a new named root atom."
#handler="""
  args match {
    case Args(SymbolLiteral(_, sym)) =>
      NamedRootType(sym.name)
    case _ => as_is
  }
"""}
```

matching in unusual ways.[5] If no type is specified for a variable, Elision assigns it the special root type `ANY`.

A variable instance may have an associated guard that controls how it may be bound. This will be explained in chapter 7. Variables may also have zero or more labels attached to them. A *label* consists of a symbol preceded by an at symbol (`@`). An example is `$x @free @strict`. Labels are used by the *map* strategy, explained in chapter 8.

Scala. Variables are created using `core.Variable(typ,name,guard,labels)`, where `typ` is the variable type, `name` is the variable name, `guard` is an optional guard (it is `true` by default), and `labels` is an optional set of labels. Thus the Elision variable `$x:INTEGER` can be created with `Variable(INTEGER,"x")`. Scala symbols are also automatically converted to variables of type `ANY`, so it is possible to pass a Scala symbol to methods expecting an atom. Thus `$x:ANY` can also be written (depending on context) as `'x`.

3.5. Metavariables

A variable `$x` can be converted to a *meta*variable by writing it with an extra dollar sign: `$$x`. This term still denotes the variable `$x`, but prevents evaluation of operators and other applications and rewrites. A *meta-term* is any term containing at least one metavariable. Metavariables may have guards and labels, just as ordinary variables do. The variable `$x` and the metavariable `$$x` denote the same (unspecified) atom.

The function of metavariables is to delay evaluation of terms. For example, the term `is_bindable($x)` evaluates to `true` immediately, since the variable `$x` is, in fact, bindable. However the term `is_bindable($$x)` is preserved as-is until the metavariable `$$x` is rewritten to some other term. This rewriting is most easily performed using a lambda (see section §3.6).

```
e> is_bindable($x)
$_repl0 = true
e> is_bindable($$x)
$_repl1 = (is_bindable:OPREF.%($$x))
e> \$$x.is_bindable($$x)
$_repl2 = \$$‘:1‘.(is_bindable:OPREF.%($$‘:1‘))
e> \$$x.is_bindable($$x).5
$_repl3 = false
e> \$$x.is_bindable($$x).$y
$_repl4 = true
```

Metavariables are enormously useful, and should be used by convention in guards and certain other places where we want to defer evaluation.

Scala. Variables are created using `core.MetaVariable(typ,name,guard,labels)`, where `typ` is the variable type, `name` is the variable name, `guard` is the optional guard (the default is `true`), and `labels` is an optional set of labels. Thus the Elision metavariable `$$x:INTEGER` can be created with `MetaVariable(INTEGER,"x")`.

[5] This is not an attempt to be frustrating. It is a performance issue. Under the hood, bindings are stored in a hash table indexed by the interned variable name. This makes lookup faster than if we had to check for an equal name and type.

TABLE 3. Lambdas

Lambda Expression	Value
\$x.(7).12	7
\$x.$x.12	12
\$x.\$y.$x.12	\$y.12

3.6. Lambdas

A lambda expression denotes a function with a single parameter, and consists of a variable called the *lambda parameter* and an atom called the *body*. The lambda is *evaluated* (or *curried*) by applying it to another atom, called the *argument*, and then replacing instances of the lambda parameter in the body with the argument. This is all much simpler than it sounds.

Lambdas are introduced by a backslash (\) followed by the parameter (a variable), then a dot (.), and then the body (an atom). The following is a lambda that denotes a constant function whose value is 7 no matter what argument is provided.

```
\$x.7
```

The following denotes the identity function that returns the atom it is applied to.[6]

```
\$x.$x
```

The following is a lambda that creates constant functions.[7] When applied to an argument, the result is a function that always returns that argument.

```
\$x.\$y.$x
```

To apply a lambda to an atom, join the lambda and the atom with a dot (.). The lambda dot binds more tightly than the application dot. table 3 shows some examples of lambda applications and the results. The parentheses around the 7 in the first entry are required to prevent the system from seeing 7.12 as a floating point number.

Consider the last item in table 3. We move through this step-by-step to show what is going on here. First, let's parenthesize this to better see the pieces: `(\$x.(\$y.$x)).12`. Now we see that we have a lambda expression of the form `\$x.`b`.12`, where b is the lambda body. This says to replace `$x` with `12` in the body. Replacing `$x` with `12` in the body (`\$y.$x`) gives `\$y.12`, which is the answer.

If you enter `\$x.\$y.$x.12` at the Elision prompt, you will *not* see `\$y.12`. What you *will* see is `\$‘:1‘.12`. The `$y` has been replaced with the funny variable `$‘:1‘`. This variable is a DeBruijn index, and it is used to prevent lambdas from "capturing" variables. In practice you should not need to worry about DeBruijn indices; the system uses them when appropriate and keeps track of the details for you. If you are curious, please see appendix E.[8]

A lambda parameter is *matched* against the argument, and the result is used to *rewrite* the body. This is all explained in chapter 7, but in practice it means that if you specify the type of the parameter, then Elision will enforce this. Thus the lambda expression `\$x:INTEGER.$x.7` will succeed with the value 7, while the lambda expression `\$x:INTEGER.$x."Fred"` will fail with an error. This also means that the type of an atom can be extracted using a lambda. The following lambda will extract the type of the atom it is applied to.

```
\$x:$T.$T
```

[6]This is sometimes called the *I combinator*.

[7]This also has a catchy name: it is the *K combinator*.

[8]For reasons now obvious, you should avoid variable names that start with colons. Elision uses these internally, and not just for DeBruijn indices, but also for "synthetic" rules and "synthetic" parameters created during matching. Partial rule completion is explained in chapter 8 and matching is explained in chapter 7.

Thus the expression `\$x:$T.$T."Fred"` evaluates to `STRING`.

Multiple parameter lambda expressions *may* need to be parenthesized when applied. Consider the following example.

```
e> \$first.\$second.%($first,$second).(2).(1)
$_repl14 = %(2, 1)
e> \$first.(\$second.%($first,$second).2).(1)
$_repl15 = %(1, 2)
```

So what happened? The lambda dot binds more tightly than the applicative dot, and the applicative dot binds to the left, so in the first line the `2` becomes the argument to `\$first.\$second.%($first,$second)`. This is rewritten to `\$second.%(2,$second)`. Parentheses in the second line force `2` to be bound in the inner lambda first. This binding order was chosen so that arguments can, in the unparenthesized case, be given in the same order (left to right) as the parameters to which they bind.

Scala. To create a lambda use `core.Lambda(param,body)`, where `param` is the lambda parameter and `body` is the lambda body. To apply a lambda to an argument, use `core.Apply(lambda,arg)`, where `lambda` is the lambda expression and `arg` is the argument. The following shows how several lambda expressions can be created in Scala.

Lambda Expression	Scala Code
`\$x.7`	`Lambda('x,7)`
`\$x.$x`	`Lambda('x,'x)`
`\$x.\$y.$x`	`Lambda('x,Lambda('y,'x))`
`\$x:INTEGER.$x.7`	`Apply(Lambda(Variable(INTEGER,"x"),'x),7)`
`\$x:$T.$T."Fred"`	`Apply(Lambda(Variable('T,"x"),'T),"Fred")`

3.7. Lambdas and Metavariables

It is often useful to use metavariables in lambdas. In fact, it is recommended to use a metavariable for the lambda parameter. Consider `is_bindable(atom)`. This evaluates to `true` if the atom is bindable (that is, is a variable), and evaluates to `false` if it is not. Suppose I want a function to tell me whether an atom is bindable or not. I might try to write this as follows: `\$a.is_bindable($a)`. This will not work. The term `is_bindable($x)` immediately evaluates to true (since `$x` is bindable), leaving the constant lambda `\$x.true`. This is not what we want at all.

The solution is to use metavariables (see section §3.5). We write `\$$x.is_bindable($$x)`. This works as desired.

```
e> \$x.is_bindable($x).7
$_repl16 = true
e> \$$x.is_bindable($$x).7
$_repl15 = false
```

The first entry fails (because `7` is *not* bindable), but the second entry works. Once you decide to use a metavariable in one place in a lambda, however, you must use it everywhere. The reason is that the lambda variables get rewritten to De Bruijn indices (see appendix E) based on the lambda parameter. The following illustrates how this can fail.

```
e> \$x.is_bindable($$x).7
$_repl16 = true
e> \$$x.is_bindable($x).7
$_repl15 = true
```

In the first case a De Bruijn variable `$v` is created for `$x`, and the body is rewritten to `is_bindable($v)` which is true. The second case falls into the problem discussed earlier, where `is_bindable($x)` is rewritten to `true` before De Bruijn substitution can even occur.

The simple rule is to use metavariables consistently in lambdas. If you use a metavariable in the body, you must also use a metavariable for the parameter.

CHAPTER 4

Collections

Elision provides a generalized collection type called, for lack of a better name, an *atom sequence*. In its "default" form it is an ordered list of atoms, but this can be changed in a variety of ways by specifying the algebraic properties of the collection.

4.1. The Basic Ordered Sequence

A basic ordered sequence of atoms is represented by a comma-separated list of atoms, enclosed in parentheses, and prefixed with a percent sign (`%`). This "looks like" a `%` operator applied to a list of atoms. For instance the list `1` followed by `2` can be represented with `%(1,2)`. Empty sequences are permissible, too: `%()`.

The collection can be heterogeneous. For instance `%(1,"Fred")` is perfectly fine. The type of a collection is deduced by looking at the types of the contained elements. If they all have the same type T, then the type is `SEQ(`T`)`. If any has a different type, then the type of the collection is `SEQ(ANY)`.

4.2. Algebraic Properties

Algebraic properties can be assigned to the sequence, and are specified starting with a percent sign (`%`). Actually, the leading percent sign for the basic ordered sequence is itself an algebraic properties specification, albeit a very simple one. Several properties are available, and the properties can even be "detached" from the sequence, matched, and rewritten. That is, an algebraic property specification is an atom on its own. Algebraic properties specifications look like operators, and even behave a bit like operators.

Each available property is described in more detail in the following subsections. Properties are specified by a single character, which is *not* case-sensitive, and can be negated by prefixing with an exclamation mark (`!`). Properties are combined by juxtaposing them, and omitting a property altogether leaves it unspecified. The following table is a quick reference to the properties.

Properties may be given an argument in square brackets. For absorbers and identities, this will be the appropriate atom. For associativity, commutativity, and idempotency, this is typically either a Boolean literal or a variable for matching. The sense of the Boolean represents whether the property is present, so `A` alone is the same as `A[true]`, and `!A` is the same as `A[false]`.

Table 1. Algebraic Properties

Property	Character	Can Be Negated	Requires an Argument
Associativity	`A`	Yes	No
Commutativity	`C`	Yes	No
Idempotency	`I`	Yes	No
Absorber	`B`	No	Yes
Identity	`D`	No	Yes

The order of properties is not significant, and conflicts are resolved with the last property specification winning. Thus `%AA!A` is the same as `%!A`. Spaces are only permitted in the arguments to a property (inside the square brackets).

Scala. Operator properties are specified by `core.AlgProp`. This class can be used directly, but it is typically easier to use its extensions: `Associative(bool)`, `Commutative(bool)`, `Idempotent(bool)`, `Absorber(atom)`, and `Identity(atom)`. The first three take a Boolean value, and the last two take an atom. Join these with `and`. The following builds the properties specification for integer addition, which is associative, commutative, not idempotent, and has an identity of zero.

```
Associative(true) and Commutative(true) and Idempotent(false) and Identity(0)
```

There is also an extension `NoProps` that indicates no properties. This is useful if you don't care to specify any properties. Note that leaving a property unspecified is not the same as negating it; it just leaves the property unspecified.

4.2.1. Associativity. An associative collection can have its elements grouped arbitrarily, provided the subgroups have the same algebraic properties specification. Associative collections are indicated with the symbol `A`. Thus we have `%A(5,2,%A(3,4),1)` is the same as `%A(5,2,3,4,1)`. In fact, Elision will convert the former to the latter.

```
e> %A(5,2,%A(3,4),1)
$_repl0 = %A(5,2,3,4,1)
e> %A(5,2,%(3,4),1)
$_repl1 = %A(5,2,%(3,4),1)
```

In the second line the properties specifications do not match, so Elision does not flatten the elements.

4.2.2. Commutativity. A commutative collection can have its arguments re-ordered arbitrarily. Commutative collections are indicated with the symbol `C`. Thus `%C(1,2,3)` is the same as `%C(3,2,1)`. Elision will order the elements of a commutative list.

```
e> %C(5,2,%C(3,4),1)
$_repl0 = %C(1,2,5,%C(3,4))
e> %AC(5,2,%AC(3,4),1)
$_repl1 = %AC(1,2,3,4,5)
```

4.2.3. Idempotency. An idempotent collection ignores element multiplicity. This is a complicated way of saying repeated elements do not matter. Idempotent collections are indicated with the symbol `I`. Thus `%I(1,2,2,3)` is the same as `%I(1,2,3)`. In fact, Elision will discard repeated elements.

```
e> %I(1,2,2,3)
$_repl2 = %I(1,2,3)
```

4.2.4. Identity. A special element may be designated as the identity. Including the identity in the sequence does not change its meaning. In fact, Elision will discard instances of the identity from the collection. An identity is indicated by the character `D` followed by the identity element in square brackets. Thus `%D(1,0,0,3)` is the same as `%(1,3)`.

```
e> %D[0](1,0,0,3)
$_repl3 = %D[0](1,3)
```

4.2.5. Absorber. Similar to an identity, a special element may be designated as an absorber. If an absorber is present, the meaning is the same as if *only* the absorber were present. Because of this, if Elision finds an absorber it dicards everything else (including other instances of the absorber) from the collection. An absorber is indicated by the character `B` followed by the absorber element in square brackets. Thus `%B[0](1,0,0,3)` is the same as `%B[0](0)`.[1]

```
e> %D[0]B[0](1,0,0,3)
$_repl4 = %B[0]D[0](0)
```

4.3. Simple Collections

Using the algebraic properties we can construct some common collections. First, the specification `%CI` denotes a *set*: a collection for which order and multiplicity are not significant. Sets within the set are preserved; if this is not desired, use `%ACI`.

```
e> %CI(4,4,5,Red,"Joe",Blue,4,%CI(Red,Blue))
$_repl5 = %CI(Red:SYMBOL,Blue:SYMBOL,4,5,"Joe",%CI(Red:SYMBOL,Blue:SYMBOL))
e> %ACI(4,4,5,Red,"Joe",Blue,4,%ACI(Red,Blue))
$_repl6 = %ACI(Red:SYMBOL,Blue:SYMBOL,4,5,"Joe")
```

If we omit idempotency or specify *not* idempotent with `!I`, then we have a multiset or *bag*, where multiple elements are preserved (though order still does no matter).

```
e> %C(4,4,5,Red,"Joe",Blue,4,%CI(Red,Blue))
$_repl7 = %C(Red:SYMBOL,Blue:SYMBOL,4,4,4,5,"Joe",%CI(Red:SYMBOL,Blue:SYMBOL))
e> %AC(4,4,5,Red,"Joe",Blue,4,%AC(Red,Blue))
$_repl8 = %AC(Red:SYMBOL,Red:SYMBOL,Blue:SYMBOL,Blue:SYMBOL,4,4,4,5,"Joe")
```

Omitting commutativity or specifying *not* commutative with `!C` results in a *list*. If we want lists to be flattened, we can specify associativity. Likewise, if we want only the first instance of an element, we can specify idempotency.

```
e> %(4,4,5,Red,"Joe",Blue,4,%(Red,Blue))
$_repl9 = %(4,4,5,Red:SYMBOL,"Joe",Blue:SYMBOL,4,%(Red:SYMBOL,Blue:SYMBOL))
e> %A(4,4,5,Red,"Joe",Blue,4,%A(Red,Blue))
$_repl10 = %A(4,4,5,Red:SYMBOL,"Joe",Blue:SYMBOL,4,Red:SYMBOL,Blue:SYMBOL)
e> %AI(4,4,5,Red,"Joe",Blue,4,%AI(Red,Blue))
$_repl11 = %AI(4,5,Red:SYMBOL,"Joe",Blue:SYMBOL)
```

4.4. Properties Specifications as Operators

It was mentioned previously that the algebraic properties specification looks like an operator. In fact, it can function similarly to one, too. We use the applicative dot (`.`) to apply it to another collection. Applied properties (on the left) *override* the properties of the argument (on the right). For instance, if `%A` is applied to `%C`, the result is `%AC`. Likewise, if `%A` is applied to `%!AC`, the result is again `%AC`.

```
e> %A.%C
$_repl11 = %AC
e> %A.%!AC
$_repl12 = %AC
e> %AD[0].%CD[2]
$_repl13 = %ACD[0]
```

[1] Yes, you can designate the same atom as both an identity and an absorber, but why? You will get a collection consisting of exactly one instance of the absorber, as absorber processing takes precedence.

```
e> (%A.%I).%(3,%AI(3),3)
$_repl14 = %AI(3)
e> %A.%I(3,%I(3),3)
$_repl15 = %AI(3,%I(3))
e> %A.%I(3,%AI(3),3)
$_repl16 = %AI(3)
```

4.5. Long Property Specifications

A properties specification like `%AC!ID[0]` is terse almost to the point of being unreadable. If a properties specification contains no variables or other complex terms, it can be written in an alternate form, replacing the letters with the entire property name, and negation with `not`. The above specification can be written in the following form.

```
% associative, commutative, not idempotent, identity 0
```

The commas are required, and spaces are allowed. This form is much more readable, but also much longer. Note that absorbers are specified via the `absorber` keyword, followed by the absorber atom.

```
e> %associative.%commutative
$_repl1 = %AC
e> %associative.% not associative, commutative
$_repl2 = %AC
e> (%associative, identity 0).%commutative, identity 2
$_repl3 = %ACD[0]
e> (%associative.%idempotent).%(3,%associative,idempotent(3),3)
$_repl4 = %AI(3)
e> %associative.%idempotent(3,%idempotent(3),3)
$_repl5 = %AI(3,%I(3))
e> %associative.%idempotent(3,%idempotent, associative(3),3)
$_repl6 = %AI(3)
```

CHAPTER 5

Operators

In general, operators denote mappings on atoms. Elision manipulates operators symbolically, so they should be regarded as mathematical functions or symbols, and not in the computer language sense. Elision supports the following kinds of operator.

- A *symbolic* operator is simply an atom that is manipulated by the system through rewriting. A Scala *closure* can be provided to construct instances of the operator, making the operator a *native* operator. This is how arithmetic operators, for example, are implemented.
- An *immediate* or *case* operator denotes a kind of macro. One or more patterns are provided, and arguments are matched against patterns, in order. The result of the match is used to construct a new atom, which is the "value" of the operator application. Case operators provide for a kind of controlled polymorphism, and can operate in ways that a symbolic operator cannot.

Operators must be declared in the current context to be used by name. This is done via the built-in `decl` operator.[1] This can be used in two ways: Either as `decl.oper` or as `decl(oper)`, where `oper` is the operator to declare. The dot form is preferred in the system libraries.

Operator redeclaration is permissible, but results in a warning. Finally, operators are a special form (see chapter 6), so they may be specified in more than one way. In this chapter we present only one style of operator definition to simplify the discussion. Other forms are potentially more useful for matching and rewriting (see chapter 7).

```
e> setprop("autoop", false)
e> {operator #name=foo #params=%($x)}
$_repl1 = {:operator:SYMBOL { binds name -> foo:SYMBOL params -> %($x) }:}
e> decl($_repl1)
Declared operator foo.
e> decl.{operator #name=foo #params=%($x)}
WARNING: Redefining operator foo.
WARNING: Prior definition: {:operator:SYMBOL { binds name -> foo:SYMBOL params -> %($x) }:}
Declared operator foo.
e> foo:OPREF
$_repl2 = foo:OPREF
e> getop(foo:OPREF)
$_repl3 = {:operator:SYMBOL { binds name -> foo:SYMBOL params -> %($x) }:}
```

5.1. Symbolic Operators

A symbolic operator declaration must have the following elements.

- A *name*, specified as a symbol. For example, `add`.
- A *type*, which is the type of the fully-applied operator.
- A *parameter list*, which specifies both the parameter types and also the operator's algebraic properties.

[1] Don't like this? Use the "autodefine" function of the REPL. Enable it by setting the `autoop` property to `true`.

To specify a symbolic operator, give these items in the following form (order does not matter, with the exception that the `operator` keyword must come first):

```
{ operator #name=NAME #type=TYPE #params=PARAMS }
```

where `NAME` is the operator name, `TYPE` is the type of the fully-applied operator, and `PARAMS` is the parameter collection. The algebraic properties of this collection are the algebraic properties of the operator. If the `#type` is *not* declared, it is assumed to be `ANY`.

The following are some examples of common operators.

```
{ operator #name=iadd #type=INTEGER
  #params=%AC!ID[0]($x:INTEGER, $y:INTEGER) }
{ operator #name=imul #type=INTEGER
  #params=%AC!ID[1]B[0]($x:INTEGER, $y:INTEGER) }
{ operator #name=and #type=BOOLEAN
  #params=%ACID[true]B[false]($P:BOOLEAN, $Q:BOOLEAN) }
```

Scala. To create a definition of a symbolic operator, use

- `core.TypedSymbolicOperator(name, type, parameters, description, detail)`

where `name` is the operator's name (a string), `type` is the type of the fully-applied operator (an atom), `parameters` specifies both the operator's properties and the parameters and their types (it is an atom sequence), `description` is the short one-line description of the operator, and `detail` is the longer description of the operator and its use (see section §5.9). The last two (`description` and `detail`) are optional. The following code creates the two examples given previously (the `or` and `add` operators).

```
import ornl.elision.core._
// Make the Boolean inclusive or operator.
val orOp = TypedSymbolicOperator("or", BOOLEAN,
  AtomSeq(Associative(true) and Commutative(true) and Idempotent(true)
          and Absorber(true) and Identity(false),
          Variable(BOOLEAN,"P"), Variable(BOOLEAN,"Q")),
  "Compute the Boolean inclusive OR.",
  """|This operator computes the Boolean inclusive OR of its arguments.
     |All arguments must be Boolean values.""".stripMargin)
// Make the integer add operator definition.
val addOp = TypedSymbolicOperator("add", INTEGER,
  AtomSeq(Associative(true) and Commutative(true) and Idempotent(false)
          and Identity(0), Variable(INTEGER,"x"), Variable(INTEGER,"y")),
  "Compute the integer sum of the arguments.",
  """|Compute the integer sum of the arguments.""".stripMargin)
```

Operator definitions should be installed in an instance of `core.OperatorLibrary`, typically found in an instance of `core.Context`. A context is automatically created and provided by the REPL, so if you are interacting with the REPL you do not need to worry about this. Contexts and operator libraries are explained in more detail in **??**. If you have a context instance, you can install a new operator by first getting the operator library from the context with `operatorLibrary`, and then invoking `add(opdef)` on the instance, where `opdef` is the operator definition.

For instance, if `context` is an instance of `core.Context`, then the operators defined above can be installed with the following lines of code.

```
// Install definitions.
context.operatorLibrary.add(orOp)
```

```
context.operatorLibrary.add(addOp)
```

The operator library is also where you register a closure for a symbolic operator. The closure must take an instance of `core.ApplyData` and return an instance of `core.BasicAtom`. More details on this are given in appendix D. You register the native handler using the `core.OperatorLibrary.register(name, handler)` instance method, where `name` is the (string) name of the operator, and `handler` is the closure described above. The following registers a native constructor for `add`, but does use some methods not explained just yet.

```
// Define add as a symbolic operator.
context.operatorLibrary.register("add",
  (data: ApplyData) => {
    // Accumulate the integer literals found.
var lits:BigInt = 0
// Accumulate other atoms found.
var other = IndexedSeq[BasicAtom]()
// Traverse the list and divide the atoms.
data.args.foreach {
  x => x match {
    case IntegerLiteral(_, value) => lits += value
    case _ => other :+= x
  }
}
// Now add the accumulated literals to the list.
other :+= Literal(INTEGER, lits)
// Construct and return a new operator application.
Apply(data.op, AtomSeq(NoProps, other), true)
})
```

It is the job of the `core.Apply` class to create operator (and other) applications. While this is explained in more detail in section §5.4, we note some important things here.

First, `core.Apply` understands how to use the operator to process absorbers and identities, etc., and this processing is performed *first*. Thus it is possible to attempt to construct a symbolic operator application with a native constructor and never have the native constructor get invoked. For instance, `add(0)` is immediately rewritten to `0`, and the native constructor is never invoked.

Second, the native constructor may *re*-invoke `core.Apply`, as is done in the handler for `add` shown above. This can simplify the native constructor since it does not have to worry about identities, etc. In the `add` constructor, the result for `lits` might be zero. By re-invoking `core.Apply` it will get correctly processed.

This second case leads to a potential problem. The `core.Apply` processing will invoke the native constructor, and our native constructor invokes `core.Apply`. To prevent this recursion we pass `true` as the optional third argument to `core.Apply` to prevent re-invoking the native constructor. It is thus possible to write a native constructor that does some processing, lets itself get re-invoked, does more processing, etc., until it finally terminates the cycle.

The properties specified for the operator are interpreted a bit differently from how they are interpreted for collections. section §5.2 explains how they are interpreted.

Symbolic operators can be defined using a much more terse syntax. In this form the prototype is written directly, and the name, parameters, and type are pulled from it. For instance, the prototype `imul($x:INTEGER, $y:INTEGER):INTEGER` specifies the name, parameters and their types, and the type of the fully-applied operator. Note that it does *not* specify the properties of the operator.

Those are optionally specified by following the prototype with `is` and then the algebraic properties specification. Use `{!...}` to enclose the entire operator definition. The three operators given previously can be written as follows using this new syntax.

```
{! iadd($x:INTEGER, $y:INTEGER):INTEGER is %AC!ID[0] }
{! imul($x:INTEGER, $y:INTEGER):INTEGER is %AC!ID[1]B[0] }
{! and($P:BOOLEAN, $Q:BOOLEAN):BOOLEAN is %ACID[true]B[false] }
```

If you use long property specifications (see section §4.5) then you are allowed to drop the leading `%`.

```
{! iadd($x:INTEGER, $y:INTEGER):INTEGER is
   associative, commutative, not idempotent, identity 0}
{! imul($x:INTEGER, $y:INTEGER):INTEGER is
   associative, commutative, not idempotent,
   identity 1, absorber 0}
{! and($P:BOOLEAN, $Q:BOOLEAN):BOOLEAN is
   associative, commutative, idempotent,
   identity true, absorber false}
```

This is much more readable, but also much longer.

After specifying the prototype the description and details may be specified as usual. In fact, the type can be specified in the usual manner (with `#type`), if desired.

5.2. Operator Properties

As with collections, operators have associated algebraic properties; in fact, they are the same as those for sequences, and are specified in the same manner. For example, addition is associative, commutative, not idempotent, and has an identity of zero: `%AC!ID[0]`. Multiplication is associative, commutative, not idempotent, has an identity of one, and an absorber of zero: `%d[1]b[0]!ica`. Boolean logical or is associative, commutative, idempotent, has an identity of false and an absorber of true: `%ACID[false]B[true]`.

Each property is described in more detail in the following subsections as it applies to operators. There are restrictions on the properties of operators that are not present for collections (specifically, associativity is required in many cases), and these are discussed in the following subsections.

5.2.1. Associativity. An associative operator f can have its arguments grouped arbitrarily. Thus $f(a,b,c) = f(f(a,b),c) = f(a,f(b,c))$. Note that any argument could be replaced with another application, so any number of arguments is possible.

Suppose we define operator f to have prototype $f(a:A, b:B):F$. Since we can replace either a or b with another application of f, we conclude that $F \subseteq A$ and $F \subseteq B$. We further note the following, by associativity.

$$f(f(c:A, d:B), b:B) = f(c:A, f(d:B, b:B))$$

Thus it must be the case that $B \subseteq A$. By a similar argument we conclude that $A \subseteq B$, and thus $A = B$. For these reasons we require that all parameters of an associative operator have the same type, and that the parameter type must be the same as the overall operator type.

```
e> { operator #name=bad1 #type=INTEGER
 >   #params=%A($x: INTEGER, $y: STRING) }
ERROR: The operator bad1 is marked as associative, but all parameters
do not have the same type, as required: %A($x:INTEGER, $y:STRING)
e> { operator #name=bad2 #type=STRING
 >   #params=%A($x: INTEGER, $y: INTEGER) }
ERROR: The operator bad2 is marked as associative, but the parameter
type (INTEGER) is not the same as the fully-applied type (STRING).
```

An associative operator must declare *exactly* two parameters.

```
e> { operator #name=bad3 #type=INTEGER
 >   #params=%A($x: INTEGER) }
ERROR: The operator bad3 is marked as associative, but does not have
exactly two parameters, as required: %A($x:INTEGER)
```

Associativity is *required* by idempotency, any identity, and any absorber.

5.2.2. Commutativity. A commutative (or abelian) operator f can have its arguments reordered arbitrarily. Thus $f(a,b) = f(b,a)$. If we provide type information, we see the following: $f(a:A, b:B) = f(b:B, a:A)$, and we conclude that $A \subseteq B$ and $B \subseteq A$, or $A = B$. We thus require that all arguments to a commutative operator have the same type, though this need not be the type of the fully-applied operator.

```
e> { operator #name=bad #type=STRING
 >   #params=%C($x: INTEGER, $y: STRING) }
ERROR: The operator bad is marked as commutative, but all parameters
do not have the same type, as required: %C($x:INTEGER, $y:STRING)
```

A commutative operator must declare *at least* two parameters.

5.2.3. Idempotency. An idempotent operator f has the property that $f(a,a) = f(a)$, so repeated arguments are discarded. Because the length of the argument list can change, idempotent operators are required to be associative. An example of an idempotent operator is the Boolean inclusive *or*, since $\mathrm{or}(x,y,x) = \mathrm{or}(x,y)$. Idempotency imposes no additional restrictions on an operator beyond those imposed by associativity, which is required.

As an example, we implement multisets and sets using symbolic operators. A *multiset* allows repeated elements to be present.

```
e> decl.{ operator #name=mset #params=%AC($x,$y) }
Declared operator mset.
e> mset(7, fred, "James", blue, 7, blue)
$_repl18 = mset(fred:SYMBOL,blue:SYMBOL,blue:SYMBOL,7,7,"James")
```

A *set* does not allow repeated elements.

```
e> decl.{ operator #name=set #params=%ACI($x,$y) }
Declared operator set.
e> set(7, fred, "James", blue, 7, blue)
$_repl19 = set(fred:SYMBOL,blue:SYMBOL,7,"James")
```

Including idempotency in the list of properties allows a simple implementation of sets. Examples of idempotent operators are the logical *or* and *and*.

5.2.4. Identity. An operator f can have an associated named identity element e, with the property that $f(x,e) = f(e,x) = f(x) = x$. Instances of a named identity are discarded from the argument list. Because the length of the argument list can change, only associative operators can have a named identity.

Named identity elements include *zero* for addition, *one* for multiplication, *false* for Boolean inclusive *or*, and *true* for Boolean *and*.

Further, $f()$ is regarded as an alternate name for a named identity e. The type of the identity must match the parameter type.

5.2.5. Absorber. An operator f can have an associated absorber element z, with the property that $f(x,z) = f(z,x) = z$. If an absorber is present in the argument list then the entire operator application is reduced to the absorber. Elision currently requires that operators with absorbers be associative, though this is not essential for any mathematical reason.

Absorber elements include zero for multiplication, true for Boolean inclusive *or*, and false for Boolean *and*. The type of the absorber must match the parameter type.

5.2.6. Construction. Atoms are transformed when entered, based on their properties, before any other processing, including a native handler, is performed. Suppose we have the following definitions.

```
decl.{operator #name=foo
      #params=%AD[0]($x:INTEGER,$y:INTEGER)
      #type=INTEGER}
decl.{operator #name=bar
      #params=%AB[true]($x:BOOLEAN,$y:BOOLEAN)
      #type=BOOLEAN}
```

We observe the following based on these definitions.

- `foo($x:INTEGER, foo($y:INTEGER, $z:INTEGER))` becomes `foo($x:INTEGER, $y:INTEGER, $z:INTEGER)` because associative applications are "flattened."
- `foo(5,0,2)` becomes `foo(5,2)` because identities are discarded.
- `bar(true, false, $x:BOOLEAN)` becomes `true` because `true` is an absorber.
- `foo(0,0,0)` becomes `0` because discarding identities leaves `foo()`, which is equal to the identity `0`.

Once this processing is complete, the system applies any native handler (see appendix D).

5.3. Operator Applications

Once an operator is defined, it can be applied to arguments. If the arguments match the parameters then the operator is fully applied and a new atom is created called an *apply*. Assuming that `add`, `or`, and `typeof` are properly-defined operators, the following are all valid applications.

```
add(4,5,$x)      or($x,false)      typeof(6)      typeof(add($x,9))
```

Because of the properties of the `or` operator, the second term `or($x,false)` immediately reduces to `$x`. The last two reduce immediately to `INTEGER`.

Operator applications can be nested, provided types match. The following is an example.[2]

```
or(equal(add($x,21),64),equal(add($y,17),32))
```

Scala. Operator application is explained in section §5.4.

Before an operator can be applied, you must get an instance of that operator. If you have installed the operator definition in an operator library, you can retrieve it via either the `get(name)` instance method of `core.OperatorLibrary`, or via the `apply(name)` instance method, where `name` is the operator name. The `get(name)` method returns an optional `core.OperatorRef`, returning `None` if the operator is not known. The `apply(name)` method will will return an instance of `core.OperatorRef`, or throw an exception if the operator is not known.

The operator itself can be obtained from the `core.OperatorRef` instance via its `operator` field. The following are two ways to obtain the `add` operator, given a context instance `context`.

[2]Note that this example is immediately reduced to `false` independent of any bindings for `$x` and `$y` because `equal` is evaluated when constructed, and thus prior to the result being rewritten with the current bindings. To prevent this, use metavariables (see section §3.5).

```
context.operatorLibrary.get("add") match {
  case Some(opref) => // Do something with the operator opref.operator
  case None => // The operator is not known!
}
// The following throws an UndefinedOperatorException if
// add is not known.
context.operatorLibrary("add").operator
```

Of course, you probably don't need (or even want) the actual operator; a reference to the operator is sufficient.

Once you have the operator reference, you can apply it to arguments by just listing the arguments in parentheses after the reference, thus implicitly invoking the operator's (or reference's) `apply` method. You can also use the `core.Apply` class as explained in the next section. The following adds some numbers.

```
context.operatorLibrary("add")(2,2,9)
```

5.4. The Applicative Dot

Operator applications can be written in two forms: `add(4,5,$x)` and `add.%(4,5,$x)`. This second form uses the *applicative dot* and a collection to apply an operator to arguments. The two forms are equivalent; they are alternative ways to represent the same atom. The first form is actually a shorthand notation for the second. Both operators and atom lists are themselves atoms, and can be manipulated by the system. In chapter 7 the applicative dot notation will turn out to be very useful.

While operators are atoms, if you just evaluate `add` by itself Elision will usually interpret this as a symbol of type `SYMBOL`. To force Elision to look this operator up, write `add:OPREF`. The `OPREF` is not actually the type of the operator, but just a hint to force the Elision parser to look up the operator in the parser's context. If the operator is not known, an error is reported. The built in operator `getop` can be used to get the actual operator from an operator reference.

The type of an operator[3] is a *mapping* from some domain to some range, deduced from the operator prototype. This can be obtained using the lambda trick presented in section §3.6. With `add` defined in the current REPL session, try the following.

```
e> \$x:$T.$T.getop(add:OPREF)
repl0 = MAP(xx(INTEGER,INTEGER),INTEGER)
```

The type for the `add` operator is a mapping from pairs of integers to integers. The above expression parses correctly because the applicative dot binds the most weakly of all connectors; more weakly than the lambda dot, for instance, which binds more weakly than the type colon. Thus the expression is equal to the following.

```
(\$x:$T.$T).(getop(add:OPREF))
```

The applicative dot can be used to "glue" any two atoms together. The atom to the left of the dot is the *left-hand side (lhs)*, while the atom to the right of the dot is the *right-hand side (rhs)*. For instance the following are perfectly legal applications, though it is not entirely clear what the first and last examples "mean." The parentheses are used in the first example to prevent `7.` from being interpreted as a floating point number; a space between the `7` and the dot would also work.

```
(7).$x      (\$x:$T.$T).getop(add:OPREF)      add.%(5,6)      $x.$y
```

[3] There are two exceptions. The operators `MAP` and `xx` are used to represent the type of operators, so they get special treatment. Their type is `ANY`.

In chapter 6 we will introduce several other uses of the applicative dot.

Scala. To create an apply using the applicative dot, use the `core.Apply(lhs,rhs)` constructor, where `lhs` is the left-hand side, and `rhs` is the right-hand side. Here are the Scala code equivalents for the apply examples given previously.

```
// (7).$x
Apply(7,'x)
// (\$x:$T.$T).getop(add:OPREF)
Apply(Lambda(Variable('T,"x"),'T),
  context.operatorLibrary("getop")(Literal(OPREF,'add)))
// add.%(5,6)
Apply(context.operatorLibrary("add"), AtomSeq(NoProps, 5, 6))
// $x.$y
Apply('x,'y)
```

5.5. Operators, Symbols, and Naked Symbols

If you type a symbol at the REPL prompt without type, this is regarded as a "naked" symbol, and is subject to special treatment. If the naked symbol appears on the left-hand side of an apply, the system will attempt to interpret it as an operator reference, and will look it up in the context. If it is not found, it will issue an error.

```
e> fred
$_repl14 = fred:SYMBOL
e> fred(5)
ERROR: The operator fred is not known.
e> fred.james
ERROR: The operator fred is not known.
```

To overcome this explicitly give the type of the symbol. Any type will suppress this behavior, but the most common (and obvious) type is `SYMBOL`. Enclosing the symbol in parentheses only helps with the simple operator application, and not with the applicative dot.

```
e> fred:SYMBOL.james
$_repl15 = (fred:SYMBOL.james:SYMBOL)
e> (fred)(5)
$_repl16 = fred:SYMBOL
$_repl17 = 5
e> (fred).5
ERROR: The operator fred is not known.
e> (fred:SYMBOL).5
$_repl18 = (fred:SYMBOL.5)
e> fred:SYMBOL(5)
ERROR: The operator SYMBOL is not known.
```

The last line above is included just to emphasize that types are themselves atoms, and are subject to the same parsing rules. Entering `fred:SYMBOL:^TYPE(5)` does, however, yield two atoms because `^TYPE` is treated specially.

The opposite of this is to explicitly state that a symbol is an operator reference, using `OPREF`.

```
e> fred:OPREF
ERROR: The operator fred is not known.
```

You may be wondering why Elision doesn't simply treat a naked symbol as a simple symbol when there is no operator with that name, so `fred.5` would be accepted. The reason is that if an operator with name `fred` were later defined then the interpretation would be fundamentally changed (from a symbol to an operator), and the Elision design seeks to avoid that. This could "break" existing files of definitions and terms.

Scala. The special type `OPREF` is one place where the Scala API and REPL deviate. Consider the following code.

```
Apply(Literal(OPREF, 'fred), Literal(5)).toParseString
```

This yields the output "`(fred:OPREF.5)`," but this fails the "round trip" test; parsing this string results in an attempt to look up the `fred` operator in the current context.

This difficulty arises from the fact that there is no "universal" context. The REPL and the parser have a context where they can look up `fred`, but the overall API does not.

To prevent this future versions of Elision may do away with `OPREF` at the API level, and more rigorously dispose of it during parsing, but for now remember that `OPREF` is a piece of parser "magic."

5.6. Case Operators

A case operator declaration must have the following elements.

- A *name*, specified as a symbol. For example, `sum`.
- An optional *type*, which is the type of the fully-applied operator. If not specified, `ANY` is used.
- A *case list*, which specifies the alternatives for the operator.

To specify a case operator, give these items in the following form (order does not matter, with the exception that the `operator` keyword must come first):

```
{ operator #name=NAME #type=TYPE #cases CASES }
```

where `NAME` is the operator name, `TYPE` is the type of the fully-applied operator, and `CASES` is the comma-separated list of alternatives. Again, if the `#type` is *not* declared, it is assumed to be `ANY`.

A case definition defines an operator by explicitly giving the interpretation of the operator in terms of its argument. When the operator symbol appears on the left of an applicative dot (even when implicit) then each of the alternatives is considered, in order.

The alternatives can be any atom, but some atoms are special.

- If the atom is a *rewriter* (the "map pair" is the most common), then the rewrite is attempted. If it succeeds, then the result is the value of the operator application. If it fails, then the next alternative is tried.
- If the atom is an *applicable* (operators are the most common, but also algebraic property specifications), then the applicable is placed on the left hand side of the applicative dot, and the arguments on the right hand side. The result is the value of the operator and no further alternatives are considered.
- Otherwise the atom is the value of the operator.

The second and third cases are very simple, so we consider them first. The following creates an alias for the `add` operator.

```
e> decl.{operator #name='+' #cases add:OPREF}
Defined operator '+'.
e> '+'(5,4,3)
$_repl20 = 12
```

The following creates a constant function that yields the value one no matter what the arguments.

```
e> decl.{operator #name=one #cases 1}
Defined operator one.
e> one(4,5)
$_repl0 = 1
e> one("Tim",6,adam)
$_repl1 = 1
```

Rewriters will be more fully explained in chapter 7. Here we consider a simple one: the *map pair*. A map pair consists of a pattern against which the argument list is matched, a text arrow (->), and an atom specifying the value of the operator when the pattern matches the argument list. It is advisable to use metavariables on the right-hand side, and we will do that by convention.

The following is a general definition using cases.

```
{operator #name=sum #cases
          %($x: INTEGER) -> $$x,
          %($x: STRING) -> $$x,
          %AC($x: INTEGER, $y: INTEGER) -> add($$x,$$y),
          %A($x: STRING, $y: STRING) -> ($$x.$$y)}
```

This definition allows adding both integers (via `add`) and strings (via the applicative dot). Note that the lists can specify properties to facilitate proper matching. Note also that `sum(5,"Fred")` results in an error, as this does not match any case. Likewise, `sum()` does not match any case, either. Suppose we want to allow the no-argument case but to leave it as `sum()`. We add an alternative with the form `%() -> _` that matches the empty argument list. The value in this case is the underscore, which is a synonym for `ANY`, and is interpreted here to mean that the argument list should be left as it was at the start, yielding `sum()` as the final answer.

Suppose we never want an undefined case; when we do not match any of the usual cases, we want to accept and leave the operator application unchanged, perhaps for symbolic manipulation. We can use `ANY` (or the underscore) as the pattern to catch this case. Consider the following alternate definition of `sum`.

```
{operator #name=sum #cases
          %($x: INTEGER) -> $$x,
          %($x: STRING) -> $$x,
          %AC($x: INTEGER, $y: INTEGER) -> add($$x,$$y),
          %A($x: STRING, $y: STRING) -> ($$x.$$y),
          _ -> _}
```

In the last line the lone underscore (a synonym for `ANY`) matches anything. Thus `sum(5,"Fred")` falls to this case and the result is `sum:OPTYPE.%(5,"Fred")`, or just `sum(5,"Fred")`, where construction halts. Note that we cannot directly refer to `sum` inside the definition since the operator is not yet defined.

This generality allows some rather unorthodox things to occur. Consider the following.

```
{operator #name=typeof #cases $x:$T -> $$T}
```

Now we have an operator that generates the type of anything on the right of an applicative dot.[4]

```
e> typeof.5
$_repl4 = INTEGER
```

In Elision, `help` is implemented as a case operator. The definition is potentially instructive. The `help` operator takes either no arguments (the "all" case) or a single operator reference. We can write this as follows.

[4]This is not perfect! The lambda trick described elsewhere is often the best way to obtain the type of an atom.

```
decl.{operator #name=help #cases
  %($op: OPREF) -> _help_op($$op),
  _help_all: OPREF
}
```

Here `_help_op` and `_help_all` are symbolic operators that provide native handlers. The first case deals with entering an operator reference. The second case is an operator reference, which is an *applicable*, so Elision applies `_help_all` to the argument list, whatever it is. This results in some non-intuitive error messages.

```
e> help(5)
ERROR: Incorrect argument for operator _help_all at position 0: 5.
Sequences are not the same length.
```

To the end user this mentions a completely different operator, and complains about the length of the argument sequence.

Suppose instead we wanted to still perform the argument check (so we get a better error message). We can try to write it as follows.

```
decl.{operator #name=help #cases
  %($op: OPREF) -> _help_op($$op),
  %() -> _help_all()
}
```

This looks good, and the error messages are better, but now `help()` doesn't work!

```
e> help(5)
ERROR: Applied the operator help to an incorrect argument list: %(5)
e> help()
e>
```

We got *no* output. This is because the output of `_help_all()` is generated as a side-effect at construction time[5] (when the native handler is invoked). Thus `_help_all()` has already been evaluated in the above.[6]

What we really want is to somehow delay applying the operator to the argument list. This could be done with metavariables, but we want to use the empty argument list here, so it isn't obvious how to do that since no variables are directly involved. The answer is that, after Elision applies one of the cases and gets a result, the result is then rewritten to replace any instances of the variable `$_` with the entire argument list. This means we can do the following. (Note that we use the metavariable `$$_` by convention here.)

```
decl.{operator #name=help #cases
  %($op: OPREF) -> _help_op($$op),
  %() -> _help_all.$$_
}
```

This will work as expected. Note that this also means it is inadvisable to use the variable `$$_` elsewhere.

Case operators are quite powerful, and can help avoid the need for native operators in many cases. For instance the Boolean `not` can be implemented as follows.

[5]Why is it done this way? Why not generate a string value? Then this definition would work, right? Well, not really. The string would be static, and we want to see help for new operators defined *after* we have defined `help`.

[6]So, what *was* the result of evaluating `_help_all()`, and why wasn't it printed? The result was the symbol `_no_show`, which is treated specially by the REPL. Whenever the result of evaluating an atom is `_no_show` all output is suppressed. This does not work from the prompt.

```
def({operator #name=not #type=BOOLEAN #cases
  true -> false,
  false -> true,
  $x:BOOLEAN -> _,
  %(true) -> false,
  %(false) -> true,
  %($x:BOOLEAN) -> _
})
```

This works, performs correct type checking, and is more general because it is also directly applicable to Boolean values using the applicative dot.

Because case operators explicitly allow polymorphism their type cannot be correctly inferred as simply as it was for symbolic operators. For instance, what is the correct type for the `help` operator? For the fully-applied type, you have two options. You can figure out the correct type yourself and include it with `#type=`, or you can omit it and Elision will assign the operator the type `ANY`.

Scala. To create a definition of a case operator, use

- `core.CaseOperator(name, type, cases, description, detail)`

where `name` is the operator's name (a string), `type` is the type of the fully-applied operator (an atom), `cases` specifies the alternatives (it is an atom sequence), `description` is the short one-line description of the operator, and `detail` is the longer description of the operator and its use. The last two (`description` and `detail`) are optional. The following code creates the examples given previously (the `sum` operator).

```
import ornl.elision.core._
// Make the generic sum operator.
val List(x,y) = List("x","y").map(Variable(INTEGER,_))
val List(s,t) = List("s","t").map(Variable(STRING,_))
val List(mx,my) = List("x","y").map(MetaVariable(INTEGER,_))
val List(ms,mt) = List("s","t").map(MetaVariable(STRING,_))
val add = context.operatorLibrary("add")
val sumOp = CaseOperator("sum", ANY,
  AtomSeq(NoProps,
    MapPair(AtomSeq(NoProps,x), mx),
    MapPair(AtomSeq(NoProps,s), ms),
    MapPair(AtomSeq(Associative(true) and Commutative(true), x, y),
      add(mx,my)),
    MapPair(AtomSeq(Associative(true), s, t), Apply(ms, mt)),
    MapPair(ANY, ANY)),
  "Compute the generic polymorphic sum.",
  "This operator computes the polymorphic sum of its arguments.")
```

The last two strings provide documentation, as described in section §5.9. As with symbolic operators, the operator definition should be installed in an instance of `core.OperatorLibrary`, typically found in an instance of `core.Context`.

Unlike symbolic operators, case operators should not be given native handlers.

5.7. Recursion

Within an operator definition it is often desirable to express recursion. Suppose we define a relation among types, where `BYTE` is considered a restriction of `WORD`, which is considered a restriction of `DWORD`, which is finally considered a restriction of the general `BITSET`. We want an operator `is_restriction_of($T,$U)` to evaluate to `true` iff `$T` is a restriction of `$U`. We could try to write this as follows.

```
{ operator #name=is_restriction_of #type=BOOLEAN #cases
  %($T,$T) -> true,
  %(BYTE,$T) -> is_restriction_of(WORD,$$T),
  %(WORD,$T) -> is_restriction_of(DWORD,$$T),
  %(DWORD,$T) -> is_restriction_of(BITSET,$$T),
  _ -> false
}
```

The notion is that we promote types until the two arguments are equal, or we exhaust the cases. This will not parse because `is_restriction_of` is not yet defined. To get around this, we use the fact that after a case operator is applied the variable `$__` (two underscores) is rewritten to the operator. This means we can write the above as follows.

```
{operator #name=is_restriction_of #type=BOOLEAN #cases
  %($T,$T) -> true,
  %(BYTE,$T) -> $$__.%(WORD,$$T),
  %(WORD,$T) -> $$__.%(DWORD,$$T),
  %(DWORD,$T) -> $$__.%(BITSET,$$T),
  _ -> false
}
```

This works as we want.

5.8. Summary: Case Operators

We provide a quick summary of properties of case operators.

- Cases are evaluated in order.
- A case can be a strategy, such as a map pair. If so, then the strategy is applied to the argument. If it reports success, the result is the value of the operator (with substitutions). If the strategy reports failure, then subsequent cases are tried.
- A case can be an applicable, such as an operator reference or a lambda. If so, then the applicable is applied to the argument immediately and the result is the value of the operator (with substitutions). No further cases are tried.
- A case can be any other atom, such as a literal. If so, then that atom is the value of the operator (with substitutions). No further cases are tried.
- The variable `$_` is replaced with the argument list.
- The variable `$__` is replaced with the operator.
- If `_` or `ANY` is the value of the operator, then the result is the operator applied to the arguments with no further expansion or modification.
- If no case matches the argument, then an error is generated.
- Case operators cannot have native handlers, but can use symbolic operators with native handlers in the cases.

It is possible to have unbounded recursion. Note that using `_` or `ANY` as the result does not cause the operator to be re-evaluated, but using `$__` likely will. A simple example is `{operator #name=recur #cases $$__.$$_}`. This simply repeats the operator indefinitely. Elision will trap this and report that it has detected unbounded recursion (or at least recursion that exhausts the stack).

Recursion can, of course, be quite useful. The following shows how one can compute powers of two. (Here we use one operator, but to be more safe, we would typically use two: `pow2` and `_pow2`. The first would be the "public" operator that takes a single argument, and the second would be a hidden operator with the cases that take two arguments. We would have to declare `_pow2` first in this case, so we could refer to it from `pow2`.)

```
e> decl.{operator #name=pow2 #type=INTEGER #cases
 > %($x: INTEGER) -> $$__.%($$x,1),
 > %(0, $y: INTEGER) -> $$y,
 > %($x: INTEGER, $y:INTEGER) -> $$__.%(add(-1,$$x),add($$y,$$y))}
Declared operator pow2.
e> pow2(4)
$_repl1 = 16
e> pow2(8)
$_repl2 = 256
e> pow2(20)
$_repl3 = 1048576
```

5.9. Documenting Operators

Operators may include two additional elements. A short one-line description of the operator can be given via `#description`, while a longer description of the operator can be given via `#detail`. If these are included, they will be used by Elision's help system. The general style is to describe the operator in abstract terms in its `#description` and not mention parameters by name. Then in the `#detail` the parameters may be mentioned directly.

```
{operator #name=sum #cases
          %($x: INTEGER) -> $$x,
          %($x: STRING) -> $$x,
          %AC($x: INTEGER, $y: INTEGER) -> add($$x,$$y),
          %A($x: STRING, $y: STRING) -> ($$x.$$y),
          _ -> _
#description="Compute the generic polymorphic sum."
#detail="""Computes the polymorphic sum of arguments. The
  arguments $x and $y may be strings or integers, but must
  be the same type."""
}
```

```
e> decl.{operator #name=sum #cases
 >          %($x: INTEGER) -> $$x,
 >          %($x: STRING) -> $$x,
 >          %AC($x: INTEGER, $y: INTEGER) -> add($$x,$$y),
 >          %A($x: STRING, $y: STRING) -> ($$x.$$y),
 >          _ -> _
 > #description="Compute the generic polymorphic sum."
 > #detail="""Computes the polymorphic sum of arguments. The
 >   arguments $x and $y may be strings or integers, but must
 >   be the same type."""
 > }
Defined operator sum.
e> help(sum)
Operator: sum
```

```
Compute the generic polymorphic sum.

Prototype:
  sum . (cases) : ANY

Cases:
  (%($x:INTEGER) -> $$x)
  (%($x:STRING) -> $$x)
  (%AC($x:INTEGER,$y:INTEGER) -> (add:OPREF.%($$x, $$y)))
  (%A($x:STRING,$y:STRING) -> ($$x.$$y))
  (ANY -> ANY)

Computes the polymorphic sum of arguments. The
arguments $x and $y may be strings or integers, but must
be the same type.
```

5.10. Closures

Returning to the `pow2` example given in section §5.8, we try the following.

```
e> decl.{operator #name=pow2 #type=INTEGER #cases
 > %(0, $y: INTEGER) -> $$y,
 > %($x: INTEGER, $y: INTEGER) -> $$__.%(add(-1,$$x),add($$y,$$y))}
Declared operator pow2.
e> decl.{operator #name=pow2 #type=INTEGER #cases
 > %($x: INTEGER) -> pow2($$x,1)}
WARNING: Redefining operator pow2.
WARNING: Prior definition: {:operator:SYMBOL { binds name -> pow2:SYMBOL
type -> INTEGER cases -> %((%(0,$y:INTEGER) -> $$y),
(%($x:INTEGER,$y:INTEGER) -> ($$__.%((add:OPREF.%(-1,$$x)),
(add:OPREF.%($$y,$$y)))))) }:}
Declared operator pow2.
e> pow2(8)
$_repl1 = 256
```

Here we define the two-argument version of the operator first, then re-define the `pow2` in what looks like a recursive form. In fact what has happened is that the second definition uses the *first* definition. When Elision creates a term using an operator reference, it captures and stores the operator's definition in the term. This means that re-defining known operators will not "break" the system. In short, Elision creates a small closure for every operator reference.

This also makes it possible to "hide" partial operator definitions, as is done above. This tactic is *avoided* in the libraries provided with Elision, as the authors consider it Bad Form.

5.11. Deferring Evaluation

The Elision system has an `error` operator that can be used to abort a computation and report an error. This is potentially helpful in an operator definition, but presents a challenge. Consider the following definition.

```
decl.{operator #name=times #cases
  %($x{i_less_than($$x,0)}, $s: STRING) ->
    error("Count must be nonnegative."),
  %(0, $s: STRING) -> "",
```

```
  %($x: INTEGER, $s: STRING) -> $$__.%(add($$x,1),$s),
  %($s: STRING, $y) ->
    error("String must be second argument.")
}
```

This seems simple enough, but entering this results in the error "Count must be nonnegative." This is because the system immediately constructs and evaluates the `error` expression.

We want to defer it, but it does not contain any variables that we can turn into metavariables. Elision provides an operator, `defer`, that has the same effect. On the right-hand side of a map pair in a symbolic operator definition, `defer(OP,ARG)` remains as-is. When the right hand side is evaluated, however, `OP.ARG` is constructed.[7]

Using this, we rewrite the above and obtain the following, which works as we want.[8]

```
decl.{operator #name=times #cases
  %($s: STRING, $y) ->
    defer(error:OPREF,%("String must be second argument.")),
  %(0, $s: STRING) -> "",
  %($x{i_less_than($$x,0)}, $s: STRING) ->
    defer(error:OPREF,%("Count must be nonnegative.")),
  %($x: INTEGER, $s: STRING) -> $$__.%(add($$x,-1),$$s).$$s
}
```

[7]How is this done? Well, `defer(OP,ARG)` is rewritten to `_defer($$_,OP,ARG)`. When `$$_` is bound this becomes a term, and is rewritten to `OP.ARG`. In short, it is really just two case operators `defer` and `_defer`.

[8]This definition causes the computation to take place on the stack, and is far from the best way to do this as it is limited based on the maximum stack depth.

CHAPTER 6

Special Forms

Elision provides an atom called the *special form* that has some associated "syntactic sugar" and is widely used for many purposes. This short chapter describes this form. First, however, it is necessary to discuss a particular special form: bindings.

6.1. Bindings

In Elision, a *binding* associates a variable, such as `$x` with some atom, such as `5`. We say that `$x` is *bound* to the *value* `5`. Variables are bound by name only; Elision does not differentiate between binding `$x:INTEGER` and binding `$x:STRING`. While this has the potential to cause trouble, in practice it should not, and results in a significant performance boost. Variable bindings are written in Elision as a bindings atom, which takes the form `{binds n1->a1 n2->a2 ... }`, where `n1`, `n2`, ..., are the variable names (without `$`), and `a1`, `a2`, ..., are the respective values. The atom `{binds}` is legal, and represents the empty bindings set. Writing `{binds x->5 x->6}` is the same as `{binds x->6}`; the last binding of a variable wins.

Scala. Bindings behave like Scala `Map` objects. To create a set of bindings, write `core.Bindings(s1->a1, s2->a2, ...)`, where `s1`, `s2`, ..., are strings that specify the variable names, and `a1`, `a2`, ..., are the atoms. It is legal to specify no atoms. The following are some examples.

Elision	Scala
`{binds}`	`core.Bindings()`
`{binds x->5}`	`core.Bindings("x"->Literal(5))`
`{binds x->5 y->6}`	`core.Bindings("x"->Literal(5), "y"->Literal(6))`

It is also possible to incrementally build up a binding. The following shows an example.

```
var binds = Bindings()
binds += ("x"->5)
binds += ("y"->6)
binds.toParseString
```

This code will generate the output `{ binds x -> 5 y -> 6 }`.

Bindings are immutable, and are created and discarded by the system during matching. Thus the `core.Bindings` object is explicitly kept "lightweight," and is not itself a `core.BasicAtom`. Instead, when an atom is needed (such as when we invoke `toParseString` in the above example) an instance of `core.Bindings` is automatically transformed into an instance of `core.BindingsAtom`. The latter *is* a `core.BasicAtom`. To see this transformation you can enter the above four lines to create `binds`, and then enter `binds:BasicAtom`, or `BindingsAtom(binds)`. Likewise, an instance of `core.BindingsAtom` will be transformed into an instance of `core.Bindings` when necessary, enabling the use of iterators, etc.

6.2. Applying Bindings

Bindings are an Elision *applicable*, meaning that they behave like operators, and generate a new atom, when placed on the left-hand side of an applicative dot. When bindings are applied to an

atom, the atom is *rewritten*. Every instance of a bound variable in the atom is replaced with the bound value.

```
e> {binds x->5 y->6}.$x
$_repl0 = 5
e> {binds x->5 y->6}.($x.$y)
$_repl1 = ((5).6)
e> {binds x->5 y->6}.add($x,$y)
$_repl2 = 11
e> {binds x->5 y->6}.add($x,$y,neg($x))
$_repl3 = 6
e> {binds x->5}.{binds y->$x}
$_repl4 = { binds y -> 5 }
e> {binds x->5}.{binds y->$x}.$y
$_repl5 = 5
```

Scala. Bindings can be applied to atoms in two ways. First, they can be applied like any other applicable, using `core.Apply`. This causes the bindings to be converted to a `core.BindingsAtom`, which introduces overhead, but also prevents rewriting atoms that contain metavariables.

```
scala> Apply(Bindings("x"->Literal(5), "y"->Literal(6)),'x).toParseString
res1: String = 5
scala> Apply(Bindings("x"->Literal(5), "y"->Literal(6)),
  MetaVariable(ANY,"x")).toParseString
res2: String = ({ binds x -> 5 y -> 6 }.$$x)
```

Alternately, every atom has a `rewrite(binds)` method that accepts a set of bindings and applies the bindings to the atom, *regardless of any metavariables*. The return value of this method is a pair of the form `(atom,flag)`, where `atom` is the rewritten atom, and `flag` is true iff any rewrites were actually performed.

```
scala> MetaVariable(ANY,"x").rewrite(Bindings("x"->Literal(5),
  "y"->Literal(6)))._1.toParseString
res4: String = 5
```

In this latter case the bindings are used directly, and no `core.BindingsAtom` is constructed.

6.3. The Basic Special Form

The special form is really just an ordered pair of atoms, and can be represented as `{: atom atom :}`, where the two atoms can be anything. Because of this structure, it is easy to match and rewrite special forms (see chapter 7). The first atom of a special form is called the *tag*, and the second atom is called the *content*. The tag is subject to special interpretation when it is a symbol. In this case Elision will perform a lookup and expect the content to have a specific structure. This representation is called the *two atom form*.

```
e> {: 5 6 :}
$_repl0 = {:5 6:}
e> {: $x $y :}
$_repl1 = {:$x $y:}
e> {: fred $y :}
$_repl2 = {:fred:SYMBOL $y:}
e> {: operator $y :}
```

```
ERROR: Form operator:SYMBOL expected bindings as content, but instead found: $y.
```

Many known special forms consist of a symbol as the tag, and bindings as the content. In fact, this is common enough that there is some "syntactic sugar" for representing this combination. In fact, this alternate form, the *short form*, was used to define operators in chapter 5. An opening brace is followed by a symbol tag, then zero or more atoms, and then zero or more *bind pairs*. There are two kinds of bind pairs:

- A hash mark (`#`) followed by a symbol `s`, an equal sign, and an atom `a`, as `#s=a`. This creates the binding `s->a`.
- A hash mark (`#`) followed by a symbol `s`, a space, and then a comma-separated list of atoms `a1,a2,...,aN`, as `#s a1, a2, ..., aN`. This creates the binding `s->%(a1,a2,...,aN)`.

Any number, including zero, bind pairs may be present. A list of atoms `a1 a2 ... aN` may be present before any bind pairs. If so, then the binding ``->a1,a2,...,aN` is created.

This is really easiest to show with a series of examples.

```
e> {sara}
$_repl3 = {:sara:SYMBOL { binds }:}
e> {sara 6}
$_repl4 = {:sara:SYMBOL { binds ``->%(6) }:}
e> {sara 6 7 8}
$_repl5 = {:sara:SYMBOL { binds ``->%(6, 7, 8) }:}
e> {sara #first="Sara" #last="Felton"}
$_repl6 = {:sara:SYMBOL { binds last -> "Felton" first -> "Sara" }:}
e> {sara #first="Sara" #last="Felton" #chars Zayre, Thorina}
$_repl7 = {:sara:SYMBOL { binds last -> "Felton" chars ->
  %(Zayre:SYMBOL,Thorina:SYMBOL) first -> "Sara" }:}
e> {sara #first="Sara" #last="Felton" #chars=%(Zayre, Thorina)}
$_repl8 = {:sara:SYMBOL { binds last -> "Felton" chars ->
  %(Zayre:SYMBOL,Thorina:SYMBOL) first -> "Sara" }:}
```

So, if you want to bind `key` to a single `value`, you can use `#key=value`. If you want to bind `key` to a sequence of values, you can use `#key v1, v2, ...` Note that a binding is always created, even when it is empty, since the special form must always consist of exactly two atoms.

The purpose of the special form is to unify many different atoms that have varied structure, and provide a consistent way to represent them, to match them, and to rewrite them. As should be evident from the above example, the short form is usually much easier to enter and shorter than the two atom form. Any short form can be represented in the two atom form, but not all two atom forms can be represented in the short form.

6.4. Known Special Forms

There are several known special forms. Bindings are not treated the same as other special forms, simply because they are themselves used to define special forms. Thus the way bindings are interpreted is a bit different, as you can see from the first row of the table below. Note the additional syntactic sugar for specifying operators. This is described in section §5.1.

Scala. Creating a special form is quite easy. Use `core.SpecialForm(tag, content)`, where `tag` is the tag, and `content` is the content. Invoking this method in the companion object will check the catalog of known special forms (shown in table 1). The following code creates each of the atoms shown in the table.

TABLE 1. Known Special Forms

Tag	Meaning, Example, and "Two Atom" Form

`binds`	Interpret the content as a list of bindings.
	`{binds x->21 y->19}`
	`{:binds:SYMBOL %(x->21, y->19):}`

`map`	Map an atom as the left-hand side of an apply to every item in a collection.
	`{map add:OPREF}`
	`{:map:SYMBOL { binds ‘‘ -> %(add:OPREF) }:}`

`match`	Create an atom that matches other atoms against a pattern.
	`{match $op.$arg }`
	`{:match:SYMBOL { binds ‘‘ -> %(($op.$arg)) }:}`

`operator`	Create an operator definition.
	`{!foo($x,$y)}`
	`{operator #name=foo #params=%($x,$y)}`
	`{:operator:SYMBOL { binds name -> foo:SYMBOL params -> %($x,$y) type -> ANY }:}`

`rule`	Define a rewrite rule.
	`{rule add(neg($x),$y)->0 #rulesets MATH}`
	`{:rule:SYMBOL { binds ‘‘ -> %((add($y,neg($x)) -> 0)) rulesets -> %(MATH:SYMBOL) }:}`

```
// Create a binding. There are easier ways to do this!
val binds = SpecialForm(
  Literal('binds),
  AtomSeq(NoProps,
    MapPair(Literal('x), 21),
    MapPair(Literal('y), 19)))
// Create {map add:OPREF}.
val map = SpecialForm(
  Literal('map),
  Bindings(""->AtomSeq(NoProps, Literal(OPREF,"add"))))
// Create {match $op.$arg}.
val mat = SpecialForm(
  Literal('match),
  Bindings(""->AtomSeq(NoProps, Apply('op,'arg))))
// Create {operator #name=foo #params=%($x,$y)}.
val foo = SpecialForm(
  Literal('operator),
  Bindings("name"->Literal('foo), "params"->AtomSeq(NoProps, 'x, 'y)))
// Create {rule add(neg($x),$y)->0 #rulesets MATH}.
val List(add, neg) = List("add","neg").map(context.operatorLibrary(_))
val rule = SpecialForm(
  Literal('rule),
  Bindings(""->AtomSeq(NoProps, MapPair(add(neg('x),'y), 0)),
           "rulesets"->AtomSeq(NoProps, Literal('MATH))))
```

These examples show direct use of the `core.SpecialForm` object. Of course, each of the above has its own simpler, dedicated method of creating an instance. For example, the simpler method of creating `core.Bindings` instances is shown several times above. The following create the same special form instances.

```
// Create a binding.
val binds = Bindings("x"->21, "y"->19)
// Create {map add:OPREF}.
val map = MapStrategy(Literal(OPREF, 'add), AtomSeq(), AtomSeq())
// Create {match $op.$arg}.
val mat = MatchAtom(Apply('op, 'arg))
// Create {operator #name=foo #params=%($x,$y)}.
val foo = TypedSymbolicOperator("foo", ANY, AtomSeq(NoProps, 'x, 'y), "", "")
// Create {rule add(neg($x),$y)->0 #rulesets MATH}.
val List(add, neg) = List("add","neg").map(context.operatorLibrary(_))
val rule = RewriteRule(add(neg('x),'y), 0, Seq(), Set("MATH"))
```

There is yet one more way to create a special form. You can create and populate an instance of `core.SpecialFormHolder`. This is how the parser functions; it detects a special form, populates an instance of this object, and then passes it to an apply method that creates the appropriate special form. Since its primary use is to support parsers, it is not discussed here. Note that every special form instance, no matter how created, holds an instance of this class.

CHAPTER 7

Matching and Rewriting

Elision is a term rewriter; its primary purpose is matching and rewriting. This chapter provides an introduction to matching and rewriting, and to the components provided by Elision to perform these tasks.

7.1. Matching

Informally, a subject atom matches some pattern atom iff the pattern atom can be made equal to the subject atom, perhaps by exploiting algebraic properties or assigning to variables. More formally, the basic principle of matching in Elision is: Let s be an atom called the *subject*, and let p be an atom called the *pattern*. Then we say the subject s matches the pattern p iff there is at least one decidable set of bindings b such that $b.p = s$. There is one exception to this general rule: `ANY`. For instance, the subject `5` matches the pattern `ANY`, but there is obviously no set of bindings that can transform `ANY` into `5`, as `ANY` contains no variables.

In summary a subject matches a pattern if either the pattern is `ANY`, or if parts of the subject can be bound to variables in the pattern such that the subject is equal to the rewritten pattern. This is somewhat complicated to explain, but rather easy to show. table 1 gives some examples.

Note that it is possible to match two terms without having to bind any variables, resulting in an empty binding. As the last case in table 1 indicates, it is also possible to have many bindings. In this last case `{binds x->$a y->add($b,$c)}` and `{binds x->add($a,$b) y->$c}` are both acceptable matches, since `add` is associative. However `add` is also commutative, so additionally we permit `{binds x->$b y->add($a,$c)}`, `{binds x->$c y->add($a,$b)}`, `{binds x->add($b,$c) y->$a}`, and `{binds x->add($a,$c) y->$b}`. The outcome of a match attempt can be no match, a single match, or many matches.

Elision provides a simple way to test matching. The special form *match atom* is an applicable (so it works like an operator) that has the form `{match pattern}`, where `pattern` is the pattern to attempt to match. The following examples reproduce the matches from table 1. The result will either be a binding if the subject matches the pattern, or `NONE` if it does not. In the case of multiple matches, only the first is returned.

Table 1. Examples of Matching

Pattern	Subject	Outcome
`5`	`5`	Match with `{binds}`
`ANY`	`5`	Match with `{binds}`
`5`	`ANY`	Match with `{binds}`
`$x`	`5`	Match with `{binds x->5}`
`5`	`$x`	No matches
`$x.5`	`joe:SYMBOL.5`	Match with `{binds x->joe:SYMBOL}`
`$x.$x`	`(5).6`	No matches
`$x.$x`	`(6).6`	Match with `{binds x->6}`
`add($x,$y)`	`add($a,$b,$c)`	Multiple matches

```
e> {match 5}
$_repl0 = {:match:SYMBOL { binds `` -> %(5) }:}
e> {match 5}.5
$_repl1 = { binds  }
e> {match ANY}.5
$_repl2 = { binds  }
e> {match 5}.ANY
$_repl3 = { binds  }
e> {match $x}.5
$_repl4 = { binds x -> 5 }
e> {match 5}.$x
$_repl5 = NONE
e> {match $x.5}.(joe:SYMBOL.5)
$_repl6 = { binds x -> joe:SYMBOL }
e> {match $x.$x}.((5).6)
$_repl7 = NONE
e> {match $x.$x}.((6).6)
$_repl8 = { binds x -> 6 }
e> {match add($x,$y)}.add($a,$b,$c)
$_repl9 = { binds x -> $a y -> add($b,$c) }
```

7.2. Restrictions

The subject `add(5,$x)` can be said to match the pattern `add(3,$y)`, since `$y` could be bound to `add($x,2)`. When we make this substitution we obtain `add(3,add($x,2))`, which is `add(3,$x,2)` by associativity, and ultimately `add(5,$x)`. Elision does not allow this match; literals in the pattern must be matched by equal literals in the subject. Thus `add(3,$x)` can match the pattern, but not `add(5,$x)`.

While Elision allows grouping of associative arguments to match a pattern, it does not allow the creation of additional arguments. That is, the subject `add($x)` does not match the pattern `add($y,$z)`, even though binding `$y` to zero and `$z` to `$x` would work. You should also note that the subject `add(0,$x)` will *not* match the subject `add($y,$z)`. This is because `add(0,$x)` is transformed into `$x` at construction time, and this becomes the subject.

Elision will also *not* construct an operator instance in order to match. For example, the subject `$x` could be made to match the pattern `and($y,$z)` by noting that `$x` is another name for `and(true,$x)`.

These limitations have the beneficial effect of making Elision computationally feasible and also of making the matches more predictable.

7.3. Map Pairs

We can use the result of a match to rewrite another atom. The following is one simple example.

```
e> {match add($x,$y,neg($x))}.add($a,neg($a),19)
$_repl10 = { binds x -> $a y -> 19 }
e> {match add($x,$y,neg($x))}.add($a,neg($a),19).$y
$_repl11 = 19
```

Remember that the applicative dot is left-associative, so the bindings are created first, and then used to rewrite the atom `$y`, yielding the result `19`. Of course this trick won't work so well when `NONE` is returned. We can build a solution using lambdas, but it will be a bit tedious.

Elision provides a simpler way: the *map pair*. The map pair has the form `pattern -> newatom`, where `pattern` is the pattern to match, and `newatom` is the atom to rewrite if a match is obtained. The map pair is a *rewriter*, meaning that it yields a specialized binding of the form `{binds atom->newatom flag->theflag}`, where `newatom` is the result and `theflag` indicates whether the rewriter succeeded in some sense. For the map pair, `theflag` is true iff the pattern match succeeded.

The following is an example.

```
e> (add($x,neg($x),$y)->$y).add($a,neg($a),19)
$_repl12 = { binds flag -> true atom -> 19 }
e> (add($x,neg($x),$y)->$y).add($a,neg($a))
$_repl13 = { binds flag -> false atom -> add($a, neg($a)) }
e> (add($x,neg($x),$y)->$y).add($a,neg($a)).$atom
$_repl14 = add($a, neg($a))
```

In the first case the match succeeds. In the second, it fails. Note that when the match fails the result is the unmodified subject.

7.4. Variable Guards

Variables can be matched based on type information.

```
e> {match $x:INTEGER}."Korra"
$_repl0 = NONE
e> {match $x:INTEGER}.5
$_repl1 = { binds x -> 5 }
```

This is often sufficient, but sometimes we would like to have greater control over the matching process. For example, we might like to match only a non-zero integer. We can do this with a *variable guard*. A variable guard is an atom that is attached to a variable using braces. The atom is rewritten with the proposed variable binding and, if it evaluates to `true`, the binding succeeds. Otherwise the entire match is rejected.

```
e> {match $x{not(equal($$x,0))}:INTEGER}.5
$_repl2 = { binds x -> 5 }
e> {match $x{not(equal($$x,0))}:INTEGER}.0
$_repl3 = NONE
```

Using metavariables in the guards prevents trouble, and we adopt it as a convention. If we did not use it, `equal($x,0)` would immediately evaluate to `false`, which would be negated to `true`, making the guard `{true}`. This would always succeed, which is not what we want at all.

We can use this to gain some control when there are many possible matches. Consider the following.

```
e> decl.{! foo($x,$y) is %AC}
Defined operator foo.
e> {match foo($x:INTEGER, $y)}.foo("Jim",5,"Tim",0)
$_repl2 = { binds x -> foo(5,"Jim","Tim") y -> 0 }
e> {match foo($x{equal($$x,0)}:INTEGER, $y)}.foo("Jim",5,"Tim",0)
$_repl3 = { binds x -> 0 y -> foo("Jim", 5, "Tim") }
```

Variable guards are sufficient in most cases, but there are still cases in which you want a guard that involves several variables. How to do this will be discussed in chapter 8.

7.5. Subtypes and Supertypes

Recall the definition of `is_restriction_of` in section §5.7. That definition of `is_restriction_of` allows us to define a variable more precisely using a variable guard and create a kind of type hierarchy. We can now write `$w{is_restriction_of($$w,WORD)}`, and `$w` can only be bound to a `WORD` or a "subtype" of `WORD`. Likewise `$W{is_restriction_of(WORD,$$W)}` can only be bound to a "supertype" of `WORD`.

```
e> decl.{operator #name=is_restriction_of #type=BOOLEAN #cases
 >   %($T,$T) -> true,
 >   %(BYTE,$T) -> $$__.%(WORD,$$T),
 >   %(WORD,$T) -> $$__.%(DWORD,$$T),
 >   %(DWORD,$T) -> $$__.%(BITSET,$$T),
 >   _ -> false
 > }
Declared operator is_restriction_of.
e> {match $w{is_restriction_of($$w,WORD)}}.BYTE
$_repl22 = { binds w -> BYTE:SYMBOL }
e> {match $w{is_restriction_of($$w,WORD)}}.DWORD
$_repl23 = NONE
e> {match $W{is_restriction_of(WORD,$$W)}}.WORD
$_repl19 = { binds W -> WORD:SYMBOL }
e> {match $W{is_restriction_of(WORD,$$W)}}.DWORD
$_repl20 = { binds W -> DWORD:SYMBOL }
e> {match $W{is_restriction_of(WORD,$$W)}}.BYTE
$_repl21 = NONE
```

CHAPTER 8

Rules and Strategies

Basic matching and rewriting were introduced in chapter 7. This chapter will explain how to manage rewriting on a much larger scale, creating more general rewrite rules, managing them with rulesets, and controlling the process through the use of strategies and the build-in strategy combinators.

8.1. Rewrite Rules and Guards

A rewrite rule generalizes the notion of a map pair a bit by introducing additional structures. In its simplest form a rule is *just* a map pair. Consider the atom $p \wedge (\neg p \vee q)$, where p and q are Booleans. Any atom of this form is equivalent to just q. We can express this as a map pair: `and($p, or(not($p), $q)) -> $q`. We can then apply this to more complex atoms to simplify them.

```
e> inc("bootstrap/Math.eli")
e> (and($p,or(not($p),$q)) -> $q) . and(or($x, not(or($y,$z))), or($y,$z))
$_repl1 = { binds flag -> true atom -> $x }
```

Conversion of this into a rule simply re-packages this into a special form: `{rule` *map-pair*`}`. This can be applied in the same fashion.

```
e> {rule and($p,or(not($p),$q)) -> $q} . and(or($x, not(or($y,$z))), or($y,$z))
$_repl2 = { binds flag -> true atom -> $x }
```

The Elision system provides an `eq` operator. This operator simply computes whether its two arguments are the same. There are some challenges to using this. Consider the following.

```
e> eq($x,$y)
$_repl3 = false
e> eq($x,$x)
$_repl4 = true
e> eq($x,7)
$_repl5 = false
```

The above demonstrates that `eq` is evaluated *eagerly*. Certainly when we have `eq($x,$x)` we want to reduce to `true`, but when we have `eq($x,$y)` we might like to suspend evaluation until we know more.

Let's define an `equal` operator that behaves like this. We can try the following.

```
decl.{operator #name=equal #cases
  %($x,$x) -> true,
  _ -> _
}
```

This works correctly for `equal(7,7)`, but does *not* work as we would like for `equal(7,8)`. To address this second case we'd really like to *replace* `equal(7,8)` with `eq(7,8)`. Elision provides an operator `is_constant(`*atom*`)` that evaluates to `true` if *atom* is a constant - i.e., it contains no variables - and `false` otherwise. This suggests a solution to our problem. We would like to write `equal($x,$y) -> eq($$x,$$y)`, but *only* if `$x` and `$y` are constants. We can write this as follows using variable guards.

```
{rule equal($x{is_constant($$x)},$y{is_constant($$y)})
  -> eq($$x,$$y)}
```

Now we have something that behaves much better. We have `equal(8,8)` and `equal(and($x,$y),and($x,$y))` both rewritten to `true` (by the definition), `equal(7,8)` is rewritten to `false` by the above rule, and `equal(and($x,$y),and($x,$z))` is left as-is, since neither the definition nor the rule applies.

The rule above is still equivalent to a map pair, but it is hard to see precisely what is happening because the guards on the variables obscure the pattern. We can pull these out and write them as general guards for the rule. We do this as follows.

```
{rule equal($x,$y) -> eq($$x,$$y)
  #if is_constant($$x), is_constant($$y)}
```

This rule has something new in it: the `#if` followed by the list of atoms. These work similarly to variable guards. Once bindings are proposed for `$x` and `$y`, the guards are each rewritten with the candidate bindings and, if they all evaluate to `true`, the rule is allowed. Any number of guards may be given, including zero, and rule guards may refer to any variable present in the pattern.

8.2. Rulesets

A significant advantage of rules is that they can be declared much as operators are, and then can be automatically applied to atoms. This is managed by declaring rules to be in one or more *rulesets*. This is done by specifying `#ruleset` or `#rulesets` (synonyms) followed by a sequence of labels, each of which specifies a ruleset. By convention ruleset names are upper-case.

Prior to use a ruleset must be specified. The operator `declare(RS)` declares the ruleset `RS`. The following declares a ruleset and adds a rule to the ruleset.

```
e> declare(MATH)
Declared ruleset MATH:SYMBOL.
e> decl.{rule and($p,or(not($p),$q)) -> $q #ruleset MATH}
Declared rule.
e> decl.{rule equal($x,$y) -> eq($$x,$$y)
     #if is_constant($$x), is_constant($$y) #ruleset MATH}
Declared rule.
```

In order for the system to apply the rules in a ruleset, the ruleset must be enabled. To operator `enable(RS)` enables the ruleset `RS`, while the operator `disable(RS)` disables the ruleset `RS`.

Consider the equation $c + x = d$, where c and d are both literals. This can be reduced to $x = d - c$. If the original equation was true, then so is the rewritten one. We might like Elision to automatically transform the first equation into the second one, collecting literals on one side of the equality. We can *attempt* to express this as a rule.

8.3. Strategies

In Elision a *strategy* is an atom S that, when applied to another atom A, yields two things: a potentially new atom A' and a Boolean value f. These items are packaged in a binding of the form `{bind atom->`A' `flag->`f`}`. Any atom that satisfies this requirement is a strategy.

We have already seen examples of strategies. Map pairs and rewrite rules both satisfy this requirement, and are considered strategies. The following is a simple strategy that takes an atom and returns that atom with the flag true.

```
{ operator #name=s_noop #type=STRATEGY #cases
  $a -> {binds atom->$$a flag->true} }
```

This illustrates some points about strategies. First, by convention the names of strategy operators should start with `s_`. Second, strategies are given the type `STRATEGY`, even though the result is a binding. This distinguishes the special form of binding from the more general case.

8.4. The Map Strategy

8.5. Strategy Combinators

APPENDIX A

BasicAtom

APPENDIX B

Exceptions

Elision attempts to handle exceptions rationally, and to *never* allow the system to fail completely. Once Elision is up and the REPL is running, exceptions are trapped and an attempt is made to dispose of them appropriately. The following sections describe how Elision handles exceptions.

B.1. Elision Exceptions

Elision uses a number of exceptions to signal conditions internally. These are usually serious conditions, from the point of view of the rewriter, but do *not* represent complete failure. That is, it should be possible to continue after one of these exceptions. All such exceptions extend `core.ElisionException` and provide a human-readable string as the message.

`core.ArgumentListException`: An error was detected in the argument list for an operator. This is typically a type check error, or the wrong number of arguments, as with `add(5,"Tim")`.

`core.LambdaUnboundedRecursionException`: The stack was exhausted during evaluation of a lambda expression. This typically represents unbounded recursion.

`core.LambdaVariableMismatchException`: The argument provided to a lambda does not match the lambda parameter. This is typically a type error, as with `\$x:INTEGER.$x."Not an integer."`.

`core.NativeHandlerException`: A native handler for an operator could not be parsed, or represented incomplete input (missing a closing parenthesis or brace, for instance). For example, `{! foo() #handler=" for(atom <- args) { "}`.

`core.OperatorDefinitionException`: An error occurred in evaluating an operator definition. Most likely your properties and prototype do not agree, as with `{! foo($x:INTEGER):INTEGER is associative}`.

`core.OperatorRedefinitionException`: An attempt was made to re-define an operator, but the operator library does not allow this. Under normal circumstances (i.e., using the REPL), redefinition is allowed with a warning.

`core.SpecialFormException`: A structural error was discovered interpreting a special form. Typically this means that the content was missing an required bind pair, as with `{operator #type=INTEGER}`.

B.2. Java Exceptions and Throwables

Other exceptions are handled differently by Elision.

APPENDIX C

Reading Elision Files

Elision provides several mechanisms to read files.

- The REPL. You can pipe input to the REPL, or you can use the `read` and `read_once` commands at the prompt.
- The `read(filename: String)` method of `parse.Processor`.
- The `elc` (Elision compiler) command that reads input, and then generates the Scala code to re-create the resulting context.

These methods all work essentially the same way: input is consumed from some source, parsed into a sequence of atoms, and each atom is then handled in some appropriate manner. All this work is generalized by the `parse.Processor` class. The following sections discuss how to use these to create additional facilities.

C.1. Processor

If you want to process Elision files you should begin with `parse.Processor`. This class provides a context, along with several methods to read data.

Internally, all text is ultimately processed by the `execute(String)` method. The flow of this method is as follows.

(1) Parse the text, obtaining a sequence of `parse.AtomParser.AstNode` instances.
(2) For each node, do the following for each registered instance of `parse.Processor.Handler`, in the order registered.
   (a) Pass the current node to the `handleNode(node: AtomParser.AstNode)` method. This method must return an `Option[AtomParser.AstNode]`. If the returned value is `None`, then the node is discarded. If the returned value is `Some(node)`, then the returned node becomes the current node. A simple default implementation of `handleNode` is:
   ```
   def handleNode(node: AtomParser.AstNode) = Some(node)
   ```
   (b) Pass the node to all listening actors via `actor ! node`.
   (c) Convert the node into an atom by invoking the node's `interpret` method.
   (d) Pass any current atom to the `handleAtom(atom: BasicAtom)` method. This method is responsible for performing any specialized processing of the atom *post-construction* (as the atom was already constructed during execution of the `interpret` method in `handleAtom`), and should return an optional atom. If `None` is returned then the atom is discarded and the system proceeds to the next node. If the return value is `Some(atom)`,

TABLE 1. Methods to Read Atoms

Method	Use
`read(filename: String)`	Read the content of the specified file.
`read(stream: Source)`	Read the content of the provided stream.
`execute(text: String)`	Read from the provided string.

then the returned atom becomes the current atom. A simple default implementation of `handleAtom` is:

```
def handleAtom(atom: BasicAtom) = Some(atom)
```

(e) Pass the atom to all listening actors via `actor ! atom`.

Other events can be passed to actors. Whenever a new file is provided for processing, a `FileStart` object is sent to the actors. When processing of the file is complete, a `FileEnd` is sent. When a new stream is provided for processing, a `StreamStart` is sent to all actors. When a stream is exhausted, a `StreamEnd` is sent to all actors. The flow of messages thus looks as follows, if a valid filename is provided to the `read` method.

```
FileStart
StreamStart
AtomParser.AstNode
BasicAtom
AtomParser.AstNode
BasicAtom
AtomParser.AstNode
BasicAtom
...
AtomParser.AstNode
BasicAtom
StreamEnd
FileEnd
```

C.2. Operators

The

APPENDIX D

Native Handlers

This appendix describes how to create native handlers for symbolic operators. Native handlers cannot be created for case operators, as they don't really make sense. If you want a native handler for a case operator, create a symbolic operator, provide it with a native handler, and include it as a case. The `help` operator described in section §5.6 is such an example.

A short description of creating native handlers is given in the Scala block of 5.3. This is suitable for creating them programmatically, but Elision provides support for providing the definition more directly.

Consider the operator `mod($b,$d)` that computes the remainder when `$b` is divided by `$d`. For example, `mod(5,2)` is `1`, while `mod(4,2)` is `0`. First we give the definition of the operator, without a native handler.

```
{! mod($b: INTEGER, $d: INTEGER): INTEGER}
```

The definition specifies two parameters: `$b` and `$d`, both integers.

A native *handler* consists of a Scala closure. It's argument is an instance of `core.ApplyData`, and the handler is expected to return an instance of `core.BasicAtom`. There are several fields and methods of `core.ApplyData` that can be useful in writing a native handler.

- The field `exec` provides access to an instance of `core.Executor`. This is a class that provides an `parse(text)` method, where `text` is a string to be parsed as a sequence of atoms. The `core.Executor` instance also provides access to a `core.Context` instance, and to a `core.Console` instance.
- The field `context` is the context provided by the executor. It provides access to the operator and rule libraries and to the current bindings.
- The field `console` is the console provided by the executor. It has methods `error(msg)` and `warn(msg)` to generate errors and warnings, and `emit(msg)` and `emitln(msg)` to write arbitrary text to the console.
- The field `_no_show` is a special symbol that is used by certain interactive systems, such as the REPL, to indicate that no value should be displayed as the result of the handler. This is the common return value for operators that are executed solely for their side effect. For example, this is the value returned by the `help` operator. This should be checked by object identity, not equality, so that it is possible to use `_no_show` as a symbol normally.
- The method `as_is` indicates that the result should be the application of the operator to the argument list without any further processing. This is commonly used as the return value for an operator in cases where no special processing is required.
- The field `op` is the instance of `core.SymbolicOperator` being applied.
- The field `args` is an instance of `core.AtomSeq` that contains the arguments to the operator, in order.
- The field `binds` is an instance of `core.Bindings` that binds each parameter to its corresponding argument. For associative operators the parameters bound are synthetic, and are named `$‘:1‘`, `$‘:2‘`, ...

Remember that strings and numbers are automatically converted to atoms. To take apart the argument list, a special extractor `core.Args` is provided. This is a sequence extractor for `core.AtomSeq` instances. Given all this, we can write the native handler for `mod` as follows.

```
def modHandler(data: ApplyData) = data.args match {
  case Args(IntegerLiteral(_, b), IntegerLiteral(_, d) => b mod d
  case _ => as_is
}
```

Now if `context` is the context where the operator is defined we can register the handler as follows. Note that we convert the function into a closure and pass the closure.

```
context.operatorLibrary.register("mod", modHandler _)
```

It is possible to declare a native handler for an operator from the REPL. This is done by adding a `#handler` bind pair to the definition of the operator. The value for `#handler` must be a string literal. The string literal executes in the following context.

```
import ornl.elision.core._
def _handler(_data: ApplyData): BasicAtom = {
  import _data._
  import ApplyData._
  // The #handler value is placed here.
}
```

The result is a function whose closure is stored as the native handler for the operator. Note that the operator need not be declared in any context; it is possible to use this to execute arbitrary Scala code. Consider the following.

```
e> {! _() #handler=""" "hello world!" """}.%()
$_repl0 = "hello␣world!"
e> {! _() #handler=""" println("hello world!") ; _no_show """}.%()
hello world!
```

Returning to the closure for the `mod` operator, we can define it as follows.

```
{! mod($b: INTEGER, $d: INTEGER): INTEGER
   #description="Compute the remainder of integer division."
   #detail="""Compute the remainder when $b is divided by $d."""
   #handler="""|args match {
               |  case Args(IntegerLiteral(_, b), IntegerLiteral(_, d)) =>
               |    b mod d
               |  case _ => as_is
               |}"""
}
```

As with `#detail`, if the first character is the pipe symbol (`|`), then the margin is stripped. Note that handler does not need to refer to `_data.args`, as everything in `_data` is imported. If other classes are required, you may add an `import` statement to obtain them.

Some examples follow. First we write a "hello world" operator.

```
{! foo() #handler=""" "hello world!" """}
```

Next we create a much more complex operator that evaluates its string argument. If the argument evaluates to one or more atoms, then the last atom is returned. Alternately we could return a collection of atoms. This makes use of the executor, console, and some of the special fields (like `_no_show` and `as_is`).

```
{! evaluate($text: STRING)
   #handler="""|import ornl.elision.core._
```

```
                |args match {
                |  case Args(StringLiteral(_, text)) =>
                |    exec.parse(text) match {
                |      case exec.ParseFailure(msg) =>
                |        console.error(msg)
                |        _no_show
                |      case exec.ParseSuccess(atoms) =>
                |        if (atoms.length > 0) atoms(atoms.length-1)
                |        else _no_show
                |    }
                |  case _ => as_is
                |}"""
}
```

Finally, we will use a few language features to create a `_decl` operator that can be used to declare other operators and rules.

```
// Create the def operator.
{! _decl($item)
#handler=
"""
  args(0) match {
    case op: Operator =>
      context.operatorLibrary.add(op)
      console.emitln("Declared operator " + toESymbol(op.name) + ".")
      _no_show
    case rule: RewriteRule =>
      context.ruleLibrary.add(rule)
      console.emitln("Declared rule.")
      _no_show
    case _ =>
      as_is
  }
"""
#evenmeta=true
#description="Declare a rule or operator in the current context."
#detail=
"Given a rewrite rule, add the rewrite rule to the rule library in the ".
"current context.  Given an operator, add the operator to the operator ".
"library in the current context."
}
```

There is one field in the above that requires some explanation. The `#evenmeta=true` forces evaluation of this operator even when the argument is a metaterm. Since both operator definitions and rewrite rules typically contain metaterms, this is essential. This feature breaks the general model of "metaterms prevent evaluation," so its use is not recommended. In fact, it only exists to address cases like the above.

An interesting question is how to get `_decl` itself declared without the use of another operator. That is, how could we "bootstrap" the system? Since `_decl` is capable of declaring operators, we *could* use it to declare itself. We do this with a lambda.

```
\$op.($op.%($op)).{! _decl($item) ... }
```

This yields `_decl` applied to an argument list containing itself, and thus results in the operator declaring itself, as we want.

APPENDIX E

De Bruijn Indices

Elision converts lambdas into a representation using a form of De Bruijn indices. This appendix describes how they are implemented in Elision.

A "proper" De Bruijn index indicates the the number of binds that are in scope. Thus the lambda $\lambda x.\lambda y.x$ should correctly produce the De Bruijn term $\lambda\lambda 2$, as the body binds to the outermost lambda. Elision does not actually use proper De Bruijn indices, but uses a very similar concept somewhere in between symbolic names and De Bruijn indices.

In Elision every atom has an associated De Bruijn index. Literals and (most) variables have index zero. Other terms have De Bruijn index equal to the maximum index of their parts.

Suppose we are given a lambda term such as `\$x.add($x,$y)`. The body contains no lambdas or De Bruijn indices, so it has De Bruijn index zero. The lambda parameter also contains no lambdas or De Bruijn indices, so it also has De Bruijn index zero. Thus the maximum De Bruijn index of the parts is zero. Since we are constructing a lambda, we increment the maximum index, obtaining one. The system constructs a special variable `$‘:1‘` representing the De Bruijn index, and then rewrites lambda parameter and body with `$x -> $‘:1‘`, obtaining the result `\$‘:1‘.add($‘:1‘, $y)`.

Next consider `\$x.\$y.$x`. First `\$y.$x` is constructed, yielding `\$‘:1‘.$x`, whose Dr Bruijn index (assigned by Elision) is one. Next we construct `\$x.(\$‘:1‘.$x)`. The De Bruijn index of the parameter is zero, and of the body is one, so the De Bruijn index to use for the new expression is `$‘:2‘`, and we rewrite with `$x -> $‘:2‘`. The trick is that the rewrite actually causes us to re-build the lambdas. Rewriting the body `\$‘:1‘.$x` results in a new lambda whose body is `\$‘:2‘` with De Bruijn index of two, and thus the entire lambda constructs a new

APPENDIX F

The Elision Directory Structure

The Elision repository contains several files and folders. This appendix explains what these items are, where to find them, and where to put stuff if you want to contribute. The description here contains items that are *not* part of the built distribution, since many of these are not required by the distribution. This description also contains items that are *not* found in the repository, but are created during the build process.

F.1. In the Distribution

The folders described in the following subsections are part of the Elision distribution, created by the `dist` target in `build.xml`.

F.1.1. The Root Folder. The root of the distribution is reserved for a few files.

- The `README.txt` file that describes the distribution and how to build it and start the REPL. You should only modify this if you need to correct information or document a new build.
- The `CHANGES.txt` file that describes changes from one version to the next.
- The `LICENSE.txt` file that contains the license for Elision.
- The `build.xml` file that is used by Ant to compile the system and generate the API documentation. You should only modify this if you need to change how the system is built. This is not needed if you are just adding new source files or third-party libraries.

There are scripts present in this folder.

- The `elision.sh` script sets up the `CLASSPATH` and invokes the Elision REPL. This is a Bash script that is only useful on UNIX, Mac OS X, and Linux machines, or on Windows machines with Cygwin.[1]
- The `run.sh` script sets up the `CLASSPATH` and then starts the Scala interpreter.This is a Bash script that is only useful on UNIX, Mac OS X, and Linux machines, or on Windows machines with Cygwin.

In general do not place any other items in the root folder. Instead, put them in the appropriate subfolder.

F.1.2. The `etc` Folder. This folder is reserved for the following items.

- The `config.xml` file that provides the template for the system's runtime configuration, and the `elision_configuration.dtd` that describes the format of the configuration file. You should not need to modify either of these unless you are modifying the `ornl.elision.Version` object.
- Files that provide Elision support for editors or IDE's, such as Emacs[2], Eclipse[3], and Sublime[4]. See the `editors` folder.
- Miscellaneous files, such as properties files for use in Eclipse, if desired.

[1] Elision is developed on Mac OS X and Linux platforms, and only occasionally tested on Windows, so YMMV.
[2] `http://www.gnu.org/software/emacs/`
[3] `http://www.eclipse.org/`
[4] `http://www.sublimetext.com/2`

If you have a file or files you need to add, and have no other place to put it, this is the place it belongs.

F.1.3. The `doc` Folder. This folder contains the Elision documentation, other than the `README.txt` file. Specifically, it contains the following.

- This file, `elision.pdf`.
- The `elision.lyx` file. This is a LyX[5] source file that is converted into `elision.pdf`, and contains the user and developer documentation you are reading right now.
- The `makedoc.sh` script that will compile the API documentation without requiring you to have Ant. This file runs the Scaladoc command and puts the documentation in the `api` subfolder, the same as the `build.xml` script does. This is a Bash script, and should work on UNIX and Mac OS X, Linux, and possibly on Windows under Cygwin.
- The `index.html` file used by the Elision web site.
- If the API documentation has been built, either by running the `makedoc.sh` script or by running the Ant build and specifying the `doc` target, then there will be an `api` folder containing the API documentation. Point your browser to `doc/api/index.html` to view the documentation.

Other documentation files should be put in this folder.

F.1.4. The `lib` Folder. This folder contains third party code necessary to compile and run the system. There is a simple rule for this folder: The build and the Bash scripts will search this folder for any jar files, and will add all such files found to the class path.

If you need to use third party code in Elision, and you want to contribute back to the main distribution, you must first clear the inclusion of the third-party code with the maintainers to make sure there are no licensing conflicts, and then you should place the distribution in this folder. This allows for consistent and rational versioning. If the user were required to get and install the third party libraries, they might obtain incompatible versions.

F.1.5. The `src` Folder. This folder contains the source code for Elision, organized into folders that mirror the package hierarchy. All Elision code should be placed in a package rooted under `ornl.elision`. This is where you will place new source files, as well as any extra files required by Scaladoc. The one exception is the `src/bootstrap` folder that holds the Elision source for bootstrapping the system.

F.2. From the Build

The folders described in the following subsections are created by the build process.

F.2.1. The `bin` Folder. This folder is created by the build process and contains the compiled class files and other elements that should be in the class path. In fact, the executable jar file is created by compressing this folder and adding a manifest, as specified in the `build.xml` file. Anything that should be in the class path should be placed in this folder.

F.2.2. The `latest` Folder. This folder is created by the build process and contains the most recently built executable `elision.jar` and `elision-all.jar` files. Any jar files in the third-party library folder (`lib`) are included in these files, and `elision-all.jar` also includes some of the Scala distribution's jar files. Nothing else should be present in this folder.

[5] LyX is an editor that is backed by LaTeX. If you want to edit this document, you need LyX. Fortunately it runs on almost every platform out there. Visit `http://lyx.org` for more information and links to the latest version.

F.2.3. Date-Stamped Folder(s). Every time the system is built, it creates a folder whose name is the current date, in `YYYYMMDD` format. This folder contains two items: the same `elision.jar` file found in the `latest` folder, and a jar file containing the content of the `src` folder at the time of the build. Because `elision-all.jar` is so large, it is not included. These folders can be deleted if they are not needed. Nothing else should be present in these folders.

APPENDIX G

The Elision Coding Guide

This chapter describes the general style of coding to use for contributions to Elision.

G.1. Concepts

The following definition is due to David Parnas.

DEFINITION. One code entity A *uses* another entity B iff the correct functioning of A depends on the availability of a correct implementation of B.

The "uses" relation is not the same as "invokes." A may invoke code in B without using B in the manner described above. For example, an operating system invokes programs, but the functioning of the operating system should not depend on the correct functioning of the programs.

Problems arise when there are loops in the uses relation. That is, A uses B and B uses A. Then neither A nor B work until *both* work. This condition is also referred to as a *tangle*. These are bad because nothing works until everything works. The code becomes very brittle, and changes "ripple" through the system in ways that are hard to trace.

Rule: Avoid loops in the use hierarchy.

G.2. Architectural Rules

Rule: All code must be in packages under `ornl.elision`.

G.3. Scaladoc

In general it is recommended to follow the Scaladoc guidelines found at `http://docs.scala-lang.org/style/scaladoc.html`. We make some revisions here.

Rule: The Elision style is to use Java-style documentation comments (first asterisks align and comment starts on second line), and not the style recommended for Scaladoc (second asterisks align and comments start on first line). There is better tool support for this style at present.

```
/** This is the style from the Scaladoc page.
  * Please don't use it.  Most formatters don't
  * know what to do with it.
  */

/**
 * This is the common "Javadoc" style.  Do use it.
 */
```

Rule: Document every public and protected method and field using Scaladoc. It is not necessary to document private methods and fields, but it is probably advisable.

Rule: Documentation may be omitted when a method or field overrides or implements a method from a parent class. In some cases you may wish to replace the textual documentation, but may omit the documentation of parameters and/or the return value.

G.4. Comments

Comment your code by recording design decisions, your rationale for choices, and descriptions of how complicated code sections work. Essentially record what you will want to know when you have to understand the code five years later, or when someone new to the project has to make changes and not break anything. If there is an assumptions in your code, you should record them.

Rule: Write complete sentences. Capitalize and punctuate appropriately, and make a reasonable attempt to spell correctly.

Rule: Use single-line comments for short comments. For long explanations, use the `/*...*/` style.

Rule: Use comment blocks to delineate logical sections of large classes. These take the following form:

```
//=========================================================================
// Section description goes here.
//=========================================================================
```

Don't comment every line, and don't just re-state what the line does. Assume the person reading the code (probably you) will know how to read the code. Use comments to explain what is going on. The following commentary is useless.

```
// get atoms
val atoms = atomset.filter(!_.isBindable)
// loop over atoms
for (atom <- atoms) {
  // print atom
  println(atom.toParseString)
}
```

This commentary is just a more verbose version of the above commentary.

```
// Get the unbindable atoms.
val atoms = atomset.filter(!_.isBindable)
// Print all the atoms.
for (atom <- atoms) {
  // Print the next atom on its own line.
  println(atom.toParseString)
}
```

This is actually the recommended style. Keep it simple.

```
// Print all unbindables.
val atoms = atomset.filter(!_.isBindable)
for (atom <- atoms) {
  println(atom.toParseString)
}
```

G.5. Spacing

Rule: Use two-space increments for indentation. If possible, set your editor to use spaces instead of tab characters, and do not use tab characters.

Rule: Skip a blank line between any Scaladoc documented items, and between classes no matter whether they are documented or not.

Rule: Break lines at 80 characters (or just slightly over).

G.6. Returning

APPENDIX H

Grammar

The grammar for Elision is described in extended Backus-Naur form (EBNF) in this appendix. Terminals are indicated with text in fixed font, boxed, as `terminal`. Elision is normally case-sensitive, but some terminals are case-insensitive. These are indicated with a double box, as `terminal`. Nonterminals are indicated with italics, as *nonterminal*. Character ranges in unicode are indicated by a dash, so any single nonzero digit is `1`-`9`. Productions are indicated with ::=. Parentheses indicate grouping. If an item may appear zero or more times, this is indicated with a suffixed asterisk, as (*atom*)*. Optional items are indicated with a suffixed question mark, as (*atom*)?.

In general arbitrary white space is allowed. When no whitespace is allowed between two elements of the grammar, the elements are joined by a dot, as `#`·*atom*. The special terminal `ANY` is described at the end, and corresponds to any character other than the backslash `\`.

Though not explicitly in the grammar, comments are allowed in two forms. Single line comments starting with `//` and ending with a newline, and multi-line comments starting with `/*` and ending with `*/`.

atom-seq ::= (*atom*)*

atom ::= *first-atom* `->` *first-atom*
| *first-atom* (`.` *first-atom*)*

first-atom ::= `(` *atom* `)`
| *special-form*
| *lambda*
| *apply*
| *typed-list*
| *alg-prop*
| *variable*
| *operator-symbol*
| *literal*
| *number*
| `^TYPE`

special-form ::= *alternate-operator-def*
| `{:` *atom atom* `:}`
| `{` *atom* (*atom*)* (*sf-bound-atom*|*sf-bound-list*)* `}`

alternate-operator-def ::= `{!` *SYMBOL* `(` *atom-sequence* `)` (:*first-atom*)? (*sf-bound-atom* | *sf-bound-list*)* `}`

sf-bound-atom ::= `#`·*SYMBOL* `=` *atom*

sf-bound-list ::= `#`·*SYMBOL* (*atom*)*

lambda ::= `\` *variable* `.` *first-atom*

apply ::= *SYMBOL* `(` *atom-sequence* `)`

atom-sequence ::= *atom* (`,` *atom*)*

typed-list ::= *alg-prop* `(` *atom-sequence* `)`

alg-prop ::= *alg-prop-short*
| `%` *alg-prop-long*

alg-prop-short ::= `%`·(`A` (`[` *atom* `]`)?
| `C` (`[` *atom* `]`)?
| `I` (`[` *atom* `]`)?
| `!A`
| `!C`
| `!I`
| `B` `[` *atom* `]`
| `D` `[` *atom* `]`)*

alg-prop-long ::= (`absorber` *atom*
| `identity` *atom*
| (`not`)? `associative`
| (`not`)? `commutative`
| (`not`)? `idempotent`)*

variable ::= (`$` | `$$`) *SYMBOL* (`{` *atom* `}`)? (`:` *first-atom*)? (`@` *SYMBOL*)*

operator-symbol ::= *SYMBOL* `:` `OPTYPE`

literal ::= *SYMBOL* | *STRING*

number ::= *any-number* (`:` *first-atom*)?

any-number ::= *hex-number* | *hex-number* | *binary-number* | *decimal-number* | *octal-number*

hex-number ::= *hex-integer* (· `.` *hex-digit*)? (· `P` ·*exponent*)?

binary-number ::= *binary-integer* (· `.` ·*binary-digit*)? (·(`E` | `P`)·*exponent*)?

decimal-number ::= *decimal-integer* (· `.` ·*decimal-digit*)? (·(`E` | `P`)·*exponent*)?

octal-number ::= *octal-integer* (· `.` ·*octal-digit*)? (·(`E` | `P`)·*exponent*)?

exponent ::= (`-` | `+`)? ·*any-integer*

any-integer ::= *hex-integer* | *hex-integer* | *binary-integer* | *decimal-integer* | *octal-integer*

hex-integer ::= `0X` ·(*hex-digit*)*

binary-integer ::= `0B` ·(`0` | `1`)*

decimal-integer ::= `1`-`9` ·(*digit*)*

octal-integer ::= `0` ·(`0`-`7`)*

hex-digit ::= `A`-`F` | *digit*

SYMBOL ::= (*letter* | `_`) ·(*letter* | *digit* | `_`)*
| `‘` ·(*escape* | *ANY*)* ·`‘`

STRING ::= `"` ·(*escape* | *ANY*)* ·`"`
| `"""` ·(*ANY*)* ·`"""`

escape ::= `\"` | `\"` | `\’` | `\n` | `\r` | `\t` | `\\`

letter ::= 

digit ::= 

ANY ::= *Any character other than*  .