A Guide to JPiccola

Oscar Nierstrasz Franz Achermann
Stefan Kneubuehl

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-03-003

Version 1.1—June 21, 2003

Abstract

Piccola is small, experimental *composition language* — a language for building applications from software components implemented in another, host programming language. This document describes JPiccola, the implementation of Piccola for the Java host language.

Chapter 1 ("Piccola in a Nutshell") presents a small example that illustrates the key concepts of Piccola. Chapter 2 presents a step-by-step tutorial of JPiccola including exercises that can be carried out with JPiccola version 3.7a.

Chapter 3 presents the interfaces of the standard Piccola libraries. Chapter 4 presents the syntax and informal semantics of Piccola.

Chapter 5 outlines the history of the Piccola project, and provides an annotated guide to the Piccola publications. Chapter 6 contains a list of Frequently Asked Questions (and Answers).

CR Categories and Subject Descriptors: D.2.11[Software Engineering]: Software Architectures—*Languages*; D.3.0 [Programming Languages]: Standards; D.3.2 [Programming Languages]: Language Classifications—*Very high-level languages*

General Terms: Software Components, architectural styles.

Additional Keywords: Composition language, Java, π-calculus.

Contents

Chapter 1

Piccola In a Nutshell

This chapter gives a brief overview of Piccola, its motivation, and the key terminology. Read this to get a rough idea what Piccola is about, then step through the tutorial. After completing the tutorial, you might want to re-read this chapter to put everything into context.

What is Piccola?

Piccola is a small *composition language*.

That is, Piccola is designed to be a minimal language for composing applications from software components. A composition language is very similar to a *scripting language* — a language in which you write high-level *scripts* to direct "actors" (*i.e.*, software components) to perform a play (*i.e.*, implement an application). The differences between a composition language like Piccola and a scripting language are:

- Core Piccola provides no programming language features of its own, just very primitive composition mechanisms (*i.e.*, forms, agents and channels) that are used to build higher-level abstractions.
- Piccola depends entirely on its *host language* to get any work done. In the case of JPiccola, the host language is Java.
- Basic data types, like booleans, numbers and strings, and basic control structures, like `if/then/else` and `try/catch`, are provided by standard modules written in Piccola itself, and which typically delegate behaviour to the host language.
- The way components are composed is governed by a *compositional style*, which defines the interfaces of a set of components, the connectors you may use to compose them, and the rules that govern and restrict composition — for example, in a pipes and filters styles, a data source may be connected by a pipe to a filter, which again yields a data source.
- The style in which you compose scripts is not hard-wired in Piccola, but is defined by Piccola modules. Unlike typical scripting languages which may focus on a particular problem domain, Piccola allows you to define high-level connectors for different styles of scripts. Styles for GUI composition, dataflow composition, or any other way of composing components can be defined as Piccola modules.

Forms and services

In Piccola, *everything is a form.* A "form" is essentially a nested record, which binds labels to values. Consider, for example, the following JPiccola code:

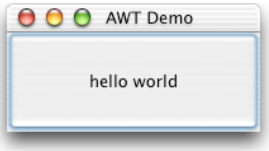

Figure 1.1: Evaluating the `helloButton` script

```
helloForm =
  text = "hello world"
  do : println text

makeFrame
  title = "AWT Demo"
  component = Button.new(helloForm) ? ActionPerformed helloForm.do
```

This code defines a form `helloForm`, which contains a binding of the label `text` to a string, and the label `do` to a service that prints the string. It then invokes a *service*, `makeFrame`, passing it a form containing bindings for the labels `title` and `component`. Indentation is significant, as it allows us to invoke `makeFrame` by passing its arguments over several lines, instead of having to write `makeFrame(title=...,component=...)`. As we shall see, `makeFrame`, `"AWT Demo"`, `200`, `println`, and all the other values here are also forms.

We use the word *form* instead of "nested record" to emphasize some important differences:

- Forms are immutable (*i.e.*, pure values), though they may wrap access to mutable entities.
- Forms may contain not only bindings of labels to values, but may also provide a *service*, allowing the form to be invoked. `makeFrame` and `helloForm.do` are both forms that provide a service, but no other bindings.
- One form may *extend* another, possible overriding bindings or services. In fact, in the example, `text = "hello world"` is extended by `do : println text`, and so on.
- A form is an *explicit namespace*, which may be used as an environment in which to evaluate a script. The lines above are evaluated in the context of a form that binds `makeFrame`, `println`, `Button` and `ActionPerformed`. The current namespace is known by the keyword `root`.

We also use the word *service* instead of "function" or "procedure" to emphasize the fact that components *provide* (and *require*) services, and that these services may be composed of other (perhaps as yet unknown) services.

Scripts and namespaces

When we evaluate this code, it generates the button we see in Figure 1.1. When we click on the button, `hello world` is printed on the Java console.

In fact, the code we have shown forms only part of a longer script that is responsible for loading a particular component composition *style*, and defining some missing glue code. The complete script is shown in Figure 1.2.

When JPiccola reads and evaluates this file, it will first set up the `root` namespace to contain some standard definitions. This is where `println` and the wrappers for Java strings and numbers are defined. The styles awt.picl and events.picl are standard styles provided by JPiccola. They wrap, respectively, Java AWT components and Java events so that we can use them from Piccola. When

```
'loadCore "awt.picl" root
'loadCore "events.picl" root
makeFrame config:
  'default=
    title="Frame"
    x=200
    y=50
    component=Label.new(text="<empty>", alignment=Label.Center)
  'config=(default,config)
  frame = Frame.new(config.title)
  frame ? WindowClosing frame.dispose
  frame.add config.component
  frame.setSize(config)
  frame.show()
helloForm =
  text = "hello world"
  do : println text
main:
  makeFrame
    title = "AWT Demo"
    component = Button.new(helloForm) ? ActionPerformed helloForm.do
```

Figure 1.2: The complete helloButton.picl script

the styles are loaded, they have access to the `root` namespace to build up their own definitions. The `helloButton` script defines a `makeFrame` glue service that will build a Java AWT Frame with a given title, size and contained component. These parameters are passed in the `config` argument form. Next, we define `helloForm`, which contains a text string and a service we would like to use. Finally, we define a binding `main`, which is a service to be invoked. `main` just invokes `makeFrame`, passing it the configuration form.

Let us now take a closer look at the `helloButton` script. Whenever a script is evaluated, it is provided with a `root` namespace (in this case, presumably the standard one that JPiccola provides). The result of evaluating a script is always a form. The `helloButton` script returns a form with a single binding for `main`, which is a service that can be evaluated. Piccola adopts the convention that, when a script is evaluated, its `main` binding, if present, will also be evaluated. If, on the other hand, the script is loaded as a style, `main` will not be evaluated. This way a Piccola file can serve as both a style and a script (cf. Python).

The two styles and the `makeFrame` service are *local* to the script. This is achieved by the quote (`'`) operator. As we shall see later, this is nothing but syntactic sugar for extending `root`.

Styles and glue

This line loads a standard JPiccola file that defines how Java AWT components are wrapped as JPiccola components:

```
'loadCore "awt.picl" root
```

The standard service `loadCore` reads the file awt.picl (which is found in a directory containing all the core modules), and evaluates it in our `root` namespace. That is to say, everything that awt.picl requires, should be provided by `root`. We then extend our own `root` namespace with the resulting form, using the quote operator.

awt.picl and events.picl are standard Piccola files defining services that wrap Java AWT components and events so that they can be accessed and composed from Piccola. awt.picl defines `Frame`

and `Button`, whereas events.picl defines `WindowClosing` and `ActionPerformed`. `loadCore`, `Host`, `false` and `println`, on the other hand, are defined by the standard prelude.picl, which is always loaded and sets up the standard `root`. `println`, for example, is a standard service that wraps the Java System.out.println method (see Section 3.1.2).

The `awt` style is rather minimal, so we need an intermediate glue service (`makeFrame`) that builds a Frame with a given title and size around a given GUI component, displays the frame, and makes sure that it will exit cleanly.

`makeFrame` is a binding in our namespace like any other binding in a form. The only difference is that it is a binding to a *service*. Ordinary bindings are defined with `=`, and services are bound with `:`.

The `awt` style allows us to bind AWT components, events and actions in a convenient way:

```
frame ? WindowClosing frame.dispose
```

binds our (wrapped Java) frame to the action `frame.dispose` in case the `WindowClosing` event is raised. Every wrapped AWT component provides a `?` connector that allows it to be bound to an event/action pair. In fact, `?` is nothing but a (Curried) service represented as an overloaded binary operator (cf. Python and C++). The result of this composition is again a component, allowing it to be composed with multiple event/action pairs.

We can use either parentheses or indentation to indicate nesting (cf. Haskell and Python). `main` invokes `makeFrame`, passing it a form containing bindings for `title`, `x`, and so on. When we invoke `Button.new`, however, we pass it the form `(text=hello)` as a parenthesized expression.

Why forms?

We close this example by noting that services are always *monadic* (*i.e.*, taking a single argument, instead of a tuple of arguments). This turns out to be the key to a lot of Piccola's flexibility. Note, for example, that `frame.setSize` is only interested in the `x` and `y` bindings of its `config` argument. It simply ignores all the other bindings. Services like `makeFrame` that require multiple arguments will normally expect them to be bundled together as a form. Nevertheless, services like the `?` may be defined as Curried abstractions, taking their arguments one at a time.

Chapter 2

A Small Piccola Tutorial

This chapter provides a step-by-step tutorial to JPiccola 3.7a including exercises. You should download the JPiccola3.7a.jar and the associated demos, and be prepared to try out the suggested exercises.

2.1 Quick Start

JPiccola requires the Java 2 Runtime Environment version 1.3.1 or better. If you don't have Java installed on your system, you can download it from java.sun.com.

Download the latest version of JPiccola from www.iam.unibe.ch/∼scg. The download includes the latest jar file, JPiccola3.7a.jar (assuming the latest version of JPiccola is 3.7a), a pdf of this tutorial, and a folder of JPiccola demo scripts. The demo file tutorial.picl contains all the tutorial examples. Note that the download does not include the JPiccola source files. These are separately available on the web site from the JPiccola cvs project.

On some systems (such as Mac OSX), you may start JPiccola just by double-clicking on the jar file. This will start up a simple Piccola console, which has been scripted in Piccola itself (see Figure 2.1). (Note that if you run JPiccola this way, you might not see any Java messages, unless you also have a Java console open.)

Alternatively you can start the JPiccola console from the command line as follows:

```
java -jar JPiccola3.7a.jar
```

You can also run a specific script either by loading it into the console and running it from there, or you can evaluate it from the command line. Suppose the file helloWorld.picl (in the Piccola demos zip) contains the following line:

```
println "hello world"
```

We can evaluate this script by running the command:

```
java -jar JPiccola3.7a.jar demo/helloWorld.picl
```

Exercise 1 *Start the JPiccola console. Load and run helloWorld.picl.*

Exercise 2 *Run helloButton.picl from the command line. (This is the example from Chapter 1; you can find it in the demos zip.) Try to load and run it from the console. Why won't it start? What must you add to run it from the console?*

2.2 Forms

In Piccola, *everything is a form.*

A form is an immutable sequence of *bindings* of labels to values:

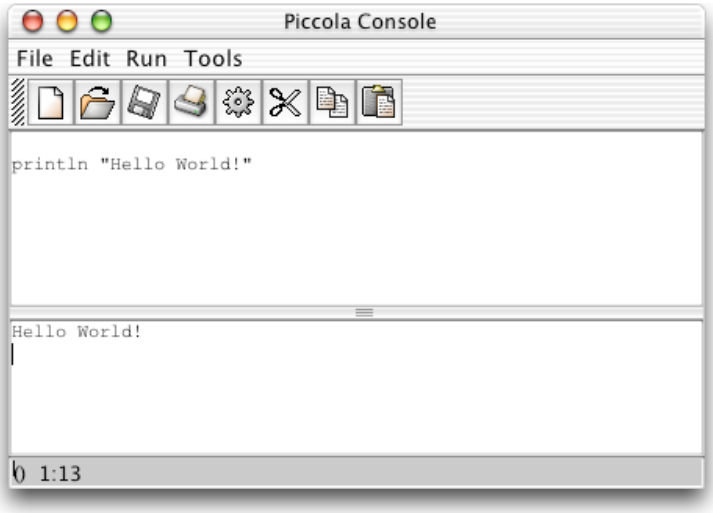

Figure 2.1: The JPiccola console

```
aPoint = (x=1, y=2)
```

This defines a form called `aPoint` with bindings for the labels `x` and `y`. The values `1` and `2` are also forms that wrap Java numbers.

Indentation can also be used to indicate nesting levels:

```
aCircle =
  centre =
    x = 3
    y = 4
  radius = 5
```

is the same as:

```
aCircle=(centre=(x=3, y=4), radius=5)
```

Please note that indented expressions must line up properly, or Piccola will not understand what you mean. The following code, for example, will generate a compilation error:

```
aCircle =
  centre = x = 3
    y = 4          # bad -- doesn't line up
   radius = 5      # bad -- doesn't line up
```

Also take care not to mix tabs and spaces when indenting!

Bindings are extracted by *projection*:

```
anotherCircle =
  centre = aCircle.centre
  radius = aCircle.radius - 1
```

This defines `anotherCircle` to be equal to `(centre=(x=3, y=4), radius=4)`.

Forms can be defined by *extending* existing forms:

```
redCircle =
  aCircle              # retrieve bindings
  radius = 1           # override a binding
  colour = "red"       # introduce a new binding
```

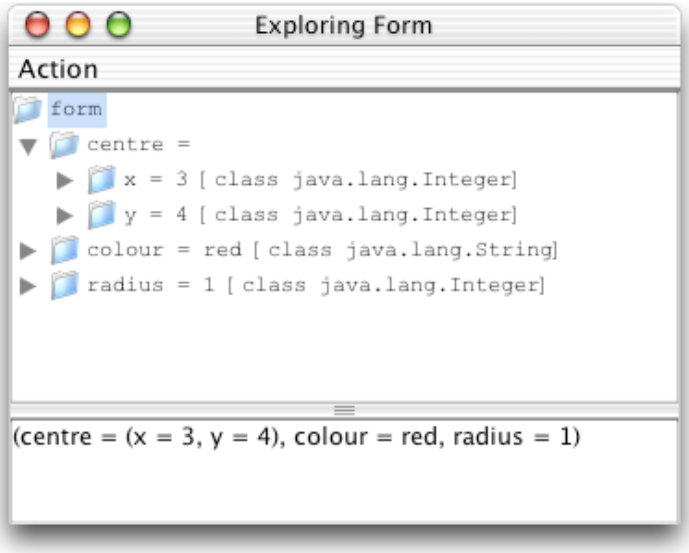

Figure 2.2: The form explorer.

`aCircle` is a form containing a set of bindings. `redCircle` *extends* these bindings with `radius=1` and `colour="red"`. The first of these *overrides* an existing binding for `radius`, whereas the second introduces a new binding.

The JPiccola console provides a simple graphical interface to explore forms. In Figure 2.2, we see the result of running:

```
explore redCircle
```

Exercise 3 *In the console, define a form* `hansel` *with bindings for* `name` *and* `address`*. Define* `gretel` *by extending* `hansel` *with a new name. Run* `explore hansel` *and* `explore gretel` *within the console. Save your script as a file,* hansel.picl*.*

2.2.1 Immutability

Bindings are immutable. A reference to an earlier binding is not affected by a later binding of the same label.

```
start =
  x = 1
  y = 2
end = start          # refers to start defined at this point
start =              # only affects the namespace below
  x = 3
  y = 4
```

Since forms are pure values, they cannot be updated. At this point, we still have `end=(x=1,y=2)`.

We can extend a form with new bindings, which may hide earlier bindings.

```
start =
  start
  x = 5
```

We may also use the "dot notation" to rebind a label in the subsequent scope:

```
start.y = 0
```

Now we have `start=(x=5,y=0)`.

Exercise 4 *Update `hansel`'s address and then explore `gretel`.*

2.2.2 The empty form

The empty form is denoted by `()`, so:

```
empty = ()
```

binds the label `empty` to an empty form.

Exercise 5 *Explore `()`. Try to project or extend `()`.*

2.2.3 Root

Everything is a form, including the current namespace (i.e., the lexical scope), which is called `root`:

```
anotherPoint =
  root.redCircle.centre          # same as: redCircle.centre
  y = root.redCircle.radius + 2  # same as: y = redCircle.radius + 2
```

Note that the current `root` changes from line to line as bindings are added or overwritten.

```
x = 1       # root.x == 1
f =
  x = 2     # root.x == 2
  y = x     # root.x == 2 and root.y == 2
# root.x == 1 and root.f=(x=2,y=2)
```

Scoping is purely *lexical* (as in most programming languages), so after `f` is defined, the bindings it has introduced for `x` and `y` are no longer visible.

Exercise 6 *Evaluate the menu item Run:Reset context. Explore `root` before and after the definitions of `hansel` and `gretel`. What does Run:Reset context do? What happens if you explore `root` again, but without running Run:Reset context?*

2.2.4 Extending `root`

Whenever we evaluate a binding *`label=expression`*, we extend both the `root` namespace as well as the *current form*. When we evaluate an expression, however, we only extend the current form.

```
blueCircle =
  redCircle          # extends blueCircle but not root
  colour = "blue"    # extends blueCircle and root
  # radius=radius+1 # would be an error!
```

The bindings for `centre`, `radius` and `colour` present in `redCircle` extend the definition of `blueCircle`, but they *do not extend its `root`* . As a consequence, it would be an error here to try to extend `blueCircle` with `radius=radius+1`, since `radius` is not defined in the `root` namespace at that point.

In order to explicitly extend `root` with the value of an expression, we may evaluate:

```
root = (root, expression)
```

Since this is a common operation, Piccola provides some syntactic sugar:

```
'expression # same as root = (root, expression)
```

Now, in the following example, the bindings of `blueCircle` are used to extend the `root` context of `greenCircle`, but they do *not* extend the definition of `greenCircle` itself.

```
greenCircle =
  'blueCircle       # extends root
  colour = "green"  # extends greenCircle and root
  radius = radius+2 # sees blueCircle.radius
  centre = centre   # sees blueCircle.centre
```

Since the bindings of `blueCircle` are not part of our `greenCircle`, we must explicitly define `centre=centre`, or our `greenCircle` will have no `centre`!

As we shall see, the quote operator is a common Piccola idiom to hide information, since it makes bindings purely local to the `root` scope of a form.

Note that by putting parentheses around a binding, we turn it into an expression, which *only* extends the current form, not `root`.

```
x = 1       # root.x == 1
f =
  'x = 0    # root.x == 0
  (x = 2)   # root.x == 0
  y = x     # root.x == 0 and root.y == 0
# root.x == 1 and root.f=(x=2,y=0)
```

We can sum up the situation as follows:

	Binding	*Expression*
Extends root	`'x=1`	`'e`
Extends current form	`(x=2)`	`e`
Extends both	`x=3`	

Exercise 7 *Extend `gretel` with `'secret="my secret"` and then explore `gretel`.*

Exercise 8 *Try to evaluate `root=()` and then explore `root`. What goes wrong?*

2.2.5 Quote and double quote

An idiom we will occasionally use is the double quote. It is used to obtain a pure side effect, ensuring that any value returned will extend *neither* `root` nor the current form:

```
''println "hello"
```

```
hello
```

Recall that `'e` evaluates `e`, extending `root` with the result of `e`. `'e`, however, returns the empty form, since its result does not extend the current form. `''e` then, evaluates `e`, and then evaluates `'()`, which extends `root` with the empty form. That is to say, it evaluates `e` and does nothing else!

If we *forget* to double quote invocations, we may accidentally extend a form without intending to!

```
john =
  name = "john"
  println "howdy! I'm " + name
```

Can we really be sure that `john` now contains only the binding for `name`, or could it be that `println` is returning an unwanted binding?

Exercise 9 *Explore `john`. Instead of invoking `println`, invoke `Debug.wrapString(name)` and explore `john` again. Note how `john` has been cluttered with other, irrelevant bindings. Now quote the invocation and explore `john` again.*

2.3 Services

Services are *abstractions over forms*. They are bound with `:` instead of `=`, and possibly introduce named arguments:

```
newPoint Args:                  # Service definition
  x = Args.x
  y = Args.y

a = newPoint aCircle.centre     # Invocation
```

gives us `a=(x=3, y=4)`.

In order to invoke a service, we *must* pass a form as an argument. If the service does not name or use its argument, we can pass the empty form:

```
hello: println "hello world"    # define a service w/o argument
hello ()                        # invoke with empty form

hello world
```

A service is also a form, so `newPoint` and `hello` are forms, and thus can be extended with bindings:

```
myHello =
  hello
  doc = "This is my very own hello form"
myHello()
println myHello.doc

hello world
This is my very own hello form
```

We can also define *Curried services*, *i.e.*, services that return services:

```
sum a b: a + b      # a Curried service
println (sum 5 6)   # need parens to parse correctly
println sum(7)(8)   # or do it this way

11
15

inc = sum 1         # bind first arg of sum -- returns a service
println inc(9)      # supply second arg

10
```

Invoking a form that does not contain a service, however, is an error:

```
a()
```

yields:

```
Exception:
Piccola Exception: ------------------------------------
Piccola Exception: Apply failed. Form is not a service.
...
```

Exercise 10 *Define a* `newCircle` *service that expects its argument to provide* `r`*,* `x` *and* `y` *bindings, and returns a form with* `radius` *and* `centre` *bindings. Explore* `newCircle(x=1,y=2,r=3)`*.*

2.3.1 Anonymous services

A service is just a binding to a lambda abstraction:

```
newPoint Args:
  x = Args.x
  y = Args.y
```

is syntactic sugar for:

```
newPoint =
  \Args:
    x = Args.x
    y = Args.y
```

(Now we can clearly see why a service is also a form: it just binds the special label \.)

The argument to a service is often a complex form, which can be passed as a subform on subsequent lines:

```
a = newPoint
  x = 5
  y = 8
```

is the same as:

```
a = newPoint(x=5, y=8)
```

Exercise 11 *Extend* `hansel` *and* `gretel` *with (anonymous) services that print their name so you can invoke* `hansel()` *or* `gretel()`*.*

2.3.2 Recursive services

Since services are just forms, they do not normally have access to their own definition. `newPoint` defined above, for example, may not call itself recursively, since the binding of `newPoint` is not in the scope of the definition. To define a recursive service, we must use the Piccola keyword `def`:

```
def newPoint Args:
  Args
  $_: "Point(x=" + Args.x + ", y=" + Args.y + ")"     # define prefix $
  _+_ other:                                          # define infix +
    newPoint
      x = Args.x + other.x
      y = Args.y + other.y
```

The service newPoint takes a single form, `Args`, as its argument. This form is wrapped and *extended* with bindings for `$` and `+`. These are, respectively, prefix and infix operators. (The arguments are represented by the `_` placeholder.) The `$` operator is special to Piccola, and is assumed by `println` to return a string representation of a form (cf. toString in Java). The `+` operator can now be invoked as an infix service of a point, and will return a new point representing the vector addition of the target point and its argument.

We must use `def` since `newPoint` is recursively defined: the body of `+` recursively invokes `newPoint`, passing it a form with bindings for `x` and `y`.

```
a = newPoint(x=5, y=6)
b = a + a
println b
```

prints:

```
Point(x=10, y=12)
```

Note that `println` prints its argument using a standard form pretty-printer, *unless* the argument provides the service `$`, in which case the result of invoking that service is printed:

```
println myHello

([service], doc = This is my very own hello form)

myHello =
  myHello
  $_: myHello.doc
println myHello

This is my very own hello form
```

Exercise 12 *Redefine `newCircle` to provide services `$` and `move`. The latter should return a new circle that is translated by the x and y positions of the point argument. Try to define `move` so it uses the `+` service provided by points.*

Exercise 13 *Extend `root` so that when you evaluate `println root`, the result is* `Hi! I'm Root!`.

2.3.3 Caveat: recursive forms

`def` may also be used to define recursive forms, but the use of recursively defined names must always be protected by a service definition, since Piccola does not support infinite forms!

```
def rForm =
  $_ : "my very own recursive form"
  \: rForm
println rForm
println rForm()
println rForm()()
```

This definition is perfectly ok, and yields:

```
my very own recursive form
my very own recursive form
my very own recursive form
```

But this is broken and will not even compile:

```
def rForm2 =
  $_ : "an infinite form"
  body = rForm2

Exception:
Piccola Exception: ------------------------------------------------------
Piccola Exception: Projection failed. Label rForm2 is not bound in form.
...
```

Exercise 14 *Explore `rForm` and `rForm()`.*

2.3.4 Default arguments

Default arguments are a common idiom in Piccola.

Our point constructor has the serious problem that it *assumes* its argument contains bindings for `x` and `y`. If we invoke `newPoint` with an *empty form* as its argument, it will fail:

```
println newPoint()
```

yields:

```
Exception:
Piccola Exception: ------------------------------------------------
Piccola Exception: Projection failed. Label x is not bound in form.
...
```

A better solution is to "pre-extend" `Args` with default values for `x` and `y`:

```
def newPoint Args:
  'Args=(x=0, y=0, Args)   # Args may override the default bindings
  Args                     # retrieve bindings with defaults
  $_: "Point(x=" + Args.x + ", y=" + Args.y + ")"
  _+_ other:
    newPoint
      x = Args.x + other.x
      y = Args.y + other.y
```

Now

```
println newPoint()
```

yields:

```
Point(x=0, y=0)
```

Note how `Args=(x=0, y=0, Args)` defines default bindings `x=0` and `y=0`, and then *extends* this form with `Arg`. When we look up `x` or `y` we are guaranteed that these labels will be bound, either to the defaults we have given, or the actual arguments, in case they have been defined. Note also that we quote the binding `'Args=(x=0, ...)` to make it purely local to the definition of `newPoint`.

Exercise 15 *Redefine* `newCircle` *to create by default a circle at the origin with radius zero. Explore* `newCircle()`*.*

2.3.5 Built-in types

Booleans, Strings and Numbers are always available as standard components wrapped from the host language.

```
n = 5 + 10 * 3 - 1
hi = "hello
world"
compare = (n == hi) & (hi == n)
println
  compare.select
    true = "the same"
    false = "not the same"
```

prints:

```
not the same
```

Note that standard Piccola components provide an interface that is independent of the interface provided by the host language. A Piccola boolean, for example, always provides a *select* service, whether or not the host Java or Squeak boolean does so.

Piccola also provides lists, sets and maps (dictionaries), which, in the case of JPiccola, are wrappers around Java List, Set and Map objects.

```
a = [1, 2, 3]
b = [4, 5, 6]
(a++b).forEach(do X: print X)
```

```
123456
```

```
println {1,2,3} ++ {2,3,4}
println {1,2,3} ++ {2,3,4} == {1,2,3,4}
```

```
{4, 1, 2, 3}
true
```

```
mymap = {{
  a -> "1 to 3"
  b -> "4 to 6"
}}
println mymap.keys()
println mymap.at(b)
```

```
[[4, 5, 6], [1, 2, 3]]
4 to 6
```

The services of Booleans, Strings and Numbers are detailed in Section 3.2.1, Section 3.2.2 and Section 3.2.4. Collections are described in Section 3.2.15.

Exercise 16 *Explore* `true`, `"hello"` *and* `1`. *Compare what you find with the descriptions in Section 3.2.*

Exercise 17 *Experiment with collections. What happens if you try to append a list to set, or vice versa?*

Exercise 18 *Explore* `"john"->"doe"`. *What is this? Experiment with the map services listed in Section 3.2.15.*

2.3.6 Language bridging

Host services can be accessed as forms. Since Java and Piccola have different calling conventions, we must somehow convert forms into something that a Java method can understand. The JPiccola/Java language bridge expects that arguments to (wrapped) Java methods are provided as JPiccola *lists* of values:

```
squareRoot N:
  Host.class("java.lang.Math").sqrt[N]
println "squareRoot(5) = " + squareRoot(5)
```

```
squareRoot(5) = 2.236067977749979
```

```
power N E:
  Host.class("java.lang.Math").pow[N, E]
println "power 2 10 = " + (power 2 10)
```

```
power 2 10 = 1024.0
```

Exercise 19 *Explore* `Host.class("java.lang.Object")`. *Can you instantiate this class?*

Exercise 20 *Explore* `Host.class("java.lang.System")`. *Can you use the resulting form to invoke* java.lang.System.out.println*? What is the difference between this and invoking the standard* `println`*? (Be sure you have the Java console open when you try this.)*

Exercise 21 *Define a service* `random` *that returns a random number.*

2.3.7 Inheritance by composition

Even though Piccola is not object-oriented, we can simulate various OO features:

```
newBetterPoint Args:
  'super = newPoint Args
  'def extend P:
    P
    _+_ other: extend P + other
    distance other:
      'dx = P.x - other.x
      'dy = P.y - other.y
      squareRoot dx*dx + dy*dy
  extend super

println newBetterPoint(x=1, y=1).distance(newBetterPoint(x=4, y=5))
```

prints:

```
5.0
```

Note how the bindings for `super` and `extend` are made private by quoting them.

Some experimental object models have been developed for Piccola that capture inheritance, as well as *self* and *super* constructs.

Exercise 22 *Define a "subclass" of* `newCircle` *that extends a circle with a distance service. (You should use the* `distance` *service of the* `newBetterPoint`*.)*

2.3.8 Control structures

Common control structures are implemented as services in prelude.picl, for example:

```
if Boolean Cases:
  'Cases = (then: (), else: (), Cases)
  Boolean.select (true = Cases.then, false = Cases.else) ()
```

Note that `if` is a Curried service. It defines default bindings for `Cases.then` and `Cases.else` which do nothing. Then it uses a Piccola boolean to select one of these services, and finally it invokes the selected service by passing it an empty form.

```
def fact N:
  if N<2
    then: 1
    else: N*fact(N-1)

println fact(5)
```

```
120
```

A `for` loop requires a form that provides `from`, `to` and `do` services:

```
hi = "good day"

for
  from: 1
  to:   hi.size()
  do i: print hi.charAt(hi.size()+1-i)
```

```
yad doog
```

We will shortly see other control structures, such as `loop` and `foreach`, but first we need variables and iterators.

Exercise 23 *What happens if you don't provide one of the `from`, `to` or `do` services expected by a `for` loop?*

Exercise 24 *Try to define `for` yourself. What problems do you run into?*

2.3.9 Dynamic scoping

The form called `dynamic` is always passed implicitly, and may be used to simulate dynamic scoping. This mechanism can be very convenient to avoid having to explicitly pass parameters that are not normally considered as being part of the interface of an object. For example, `println` actually invokes `dynamic.println` instead of the default behaviour, if it is defined. We can use this fact to modify the behaviour of `println` in services that have already bound it, like the `hello` service we defined much earlier:

```
hello()                     # calls the current println

'saveDynamic = dynamic      # save the dynamic namespace
'dynamic.println X:         # extend dynamic with our println
  'dynamic = saveDynamic    # locally restore the saved dynamic
  println ">>" + X + "<<"   # call the real println

hello()                     # println now calls our dynamic println

'dynamic = saveDynamic      # restores the old println
hello()                     # works again as it used to
```

```
hello world
>>hello world<<
hello world
```

The JPiccola console uses this feature to redirect `println` requests to the console window.

Exercise 25 *Within the console, set `dynamic` to the empty form and then try to invoke `println`. Can you explain what happens?*

Exercise 26 *Explore `dynamic`. Define a service `exploreDynamic` that explores `dynamic` and invoke it. Now extend `dynamic` and again invoke the `exploreDynamic` service you defined earlier. What do you observe? What happens if you do the same experiment with a service `exploreHansel`?*

2.4 Concurrency

Since forms are immutable, the language we have seen so far is purely functional. The only side-effects that may occur are in the host language. Piccola, however, also supports concurrent *agents*, which may communicate by means of shared *channels*.

2.4.1 Agents

You can start a concurrent agent by invoking `run`, passing it a form with a `do` service:

```
run (do: [1,2,3].forEach(do X: print X))
run (do: [4,5,6].forEach(do X: print X))
```

Might yield:

```
124536
```

Exercise 27 *Run the example a few times to see if you always get the same result. Define a* `sleep` *service that invokes* `Host.class("java.lang.Thread").sleep[N.asLong()]`*. Now insert* `sleep 1` *before or after the* `print` *and try again.*

2.4.2 Channels

Agents can communicate through shared channels, which provide `send` and `receive` services.

```
c = newChannel()
run
  do: println c.receive()
run
  do: c.send("hello from another world")
```

Yields:

```
hello from another world
```

This service will stop an agent:

```
stop: newChannel().receive()
```

Since no other agent has access to the new channel, nothing will ever be written on it, and the reader will block forever.

Exercise 28 *Define a* producer *agent that sends the numbers 1 to 10 to a channel, and a* consumer *agent that reads each number from the channel and prints it.*

Exercise 29 *Implement a* `newSemaphore` *service that creates a* semaphore *with* `p` *and* `v` *services. Use a semaphore to give two agents mutually exclusive access to shared critical section.*

2.4.3 Variables

We can use channels to model side effects by storing a value to be retrieved later:

```
newVar X:
  'var = newChannel()
  ''var.send X            # store the initial value
  set X:
    ''var.receive()       # consume the old value
    ''var.send X          # and store the new one
    X                     # return the new value
  get:
    'X = var.receive()    # read the old value
    ''var.send X          # put it back again
    X                     # now return it
  *_ = get                # prefix *
  _<-_ = set              # infix <-
```

(Actually, the standard `newVar` service provided by the prelude.picl is defined somewhat differently, but this definition is equivalent.)

Now we can use the `loop` service defined in prelude.picl:

```
x = newVar(10)

loop
  while: *x > 0
  do:
    x <- *x - 1
    print (*x)
println ""
```

```
9876543210
```

(Be careful not to type `println *x` — Piccola will assume that you are trying to multiply `println` by `x`!)

Exercise 30 *Try running the loop with a label binding `x=10` instead of a variable. (You will have to test `x>0`, rebind `x=x-1` and `print x`.) Can you explain what happens?*

Exercise 31 *Can you define a `loop` service that behaves the same as the standard one?*

2.5 Introspection

First-class labels make Piccola primitives available as services:[1]

```
centreLabel = newLabel("centre")
println centreLabel.project(aCircle)
```

```
(x = 3, y = 4)
```

The `restrict` service of a first-class label can be used to remove that label from a form:

```
println centreLabel.restrict(aCircle)
```

```
(radius = 5)
```

and `bind` is used to bind it:

```
println centreLabel.bind(x=1, y=2)
```

```
(centre = (x = 1, y = 2))
```

We can print all the labels of a form:

```
printLabels aCircle
```

```
centre, radius
```

or we can iterate over them:

```
forEachLabel
  form = aCircle
  do X:
    println X.name() + " = " + X.project(form)
```

[1] Recall that `aCircle=(centre=(x=3, y=4), radius=5)`

```
centre = (x = 3, y = 4)
radius = 5
```

These standard services are built up using the `inspect` service, which offers a finer degree of control. Here we use `inspect` to define a generic wrapper that invokes `Block` at the beginning of every service bound to some label:

```
def wrapServices Form Result Block:
  inspect Form
    isLabel L:    # select some first-class label
      'value = L.project(Form)
      'wrappedValue = if isService(value)
        then: \X: (''Block.do L, value X)
        else: value
      'Result = (Result, L.bind(wrappedValue))
      wrapServices L.restrict(Form) Result Block
    isEmpty:   Result
    isPeer:    Result
    isService: (Result, Form)

p = wrapServices newPoint() () (do L: println "" + L.name() + " invoked")
z = p + p
```

```
_+_ invoked
```

(The *peer* is the unwrapped Host entity accessible from the Piccola form that wraps it.)

We can also obtain a *meta* representation of any form by invoking the standard `meta` service. `meta(`*form*`)` allows to explore and manipulate the bindings and services of an arbitrary form.

```
foreach meta(newChannel()).labels()
  do X: println X
```

```
receive
send
```

The standard `foreach` service is a Curried service that takes an iterator and a `do` service and applies the service to each item provided by the iterator. An iterator must provide services `next`, which returns the next item, and `hasNext`, which answers `false` when there are no more items to retrieve.

For further details, see Section 3.2.7 and Section 3.2.8.

Exercise 32 *Write a recursive service that pretty-prints a form, suitably indenting each of its bindings according to its nesting level.*

2.6 Evaluating scripts

Scripts can be evaluated as strings within a given root:

```
aScript = """println "hello
world" """

eval aScript root
```

```
hello
world
```

Exercise 33 *Define a service that evaluates a string in a* `root` *that provides only* `println` *and nothing else.*

2.6.1 Loading scripts

A file may contain a set of bindings to be loaded: Suppose the file hello.picl contains:

```
hiThere = "hello from hello.picl"
hello: println hiThere
```

We may now use the standard service `loadRelative` to load and run this script:

```
'loadRelative "hello.picl" root
hello()

hello from hello.picl
```

The standard services `load` and `loadCore` load, respectively, files in arbitrary locations, and files relative to the standard prelude.

Exercise 34 *Use `loadCore` to load ide.picl with the current `root`, and invoke its `main()` service.*

Exercise 35 *Write a script that can be invoked from the command line that will start an explorer on `root`. (Hint: `explore` is defined in the core library ide.picl.)*

2.7 Testing

Piccola provides standard services to support testing and exception-handling.

2.7.1 Exceptions

Exceptions are raised using the `raise` service. Although any form can be given as an argument to `raise`, normally a string describing the error is passed.

```
raise "my error"

Exception:
Piccola Exception: my error

raise "my error" at line 2, column 1
...
```

Alternatively, you may pass a form that binds `msg=`*String*.

To catch exceptions, use the `try/catch` clause:

```
try
  do: raise(msg="eek!")
  catch e: println "caught exception: " + e.msg

caught exception: eek!
```

Exercise 36 *Raise various exceptions by performing illegal Piccola or Java operations. Catch the exceptions and explore them.*

Exercise 37 *How would you implement your own version of `try`?*

2.7.2 Assertions

Piccola provides an `assert` service that can be used to define pre- and post-conditions for services:

```
def fact N:
  assert N>=0
  if N==0
    then: 1
    else: N*fact(N-1)

try
  do:
    println fact(5)
    println fact(-1)
  catch e:
    println "Caught exception: " + e.msg
```

```
120
ASSERTION FAILED
```

Exercise 38 *Implement your own version of* `assert`*.*

2.7.3 Tests

Piccola provides a simple unit testing framework similar to JUnit (for Java) and SUnit (for Smalltalk). `PiUnit` is a drastically stripped-down version that adopts the same principles as its more sophisticated relatives:

- A single test case is constructed with `PiUnit.newTest`, which requires a `name` (a *String*) and a `test` service.
- The `test` service performs various actions, and tests the results with `PiUnit.assert`, `PiUnit.assertEquals`, `PiUnit.assertFails` and `PiUnit.assertFalse`.
- A test *suite* is constructed by composing test cases with the `+` operator.
- A test case or suite `myTest` can be run by invoking `myTest.test()` or `myTest.test(verbose=true)`.
- Tests run silently, unless the `verbose` flag is set.
- When running a test suite, each individual test case is run, and errors are reported only for tests that fail.

```
'PiUnit                          # Load the PiUnit services into root
testFact = newTest               # Define a test case
  name = "Factorial test"
  test: assert fact(5) == 120
testBetterPoint = newTest        # Another test case
  name = "BetterPoint test"
  test:
    'a = newBetterPoint(x=0,y=4)  # Build the "fixture" (test data)
    'b = newBetterPoint(x=3,y=0)
    assertEquals
      get = a.distance(b)
      expect = 5.asDouble()
myTestSuite = testFact + testBetterPoint     # Compose a test suite
myTestSuite.test(verbose=true)               # Run all the tests
```

```
Testing Factorial test...
Testing BetterPoint test...
```

Exercise 39 *Define a test suite that exercises agents and channels.*

2.8 Wrapping host entities

We have seen how JPiccola automatically wraps Java Booleans, Strings, and so on. But how can we implement our own wrappers?

Consider our old factorial function. If we evaluate:

```
println "fact(20) = " + fact(20)
```

we get a nasty surprise:

```
fact(20) = -2102132736
```

The problem is that our plain Java integers overflow. Luckily Java provides a nice class, java.math.BigInteger, that would be perfect for computing large factorials, but this class has an ugly interface, due to the design decision (*i.e.*, "mistake") to not support overloaded operators in Java. We would like to automatically *wrap* instances of java.math.BigInteger so that they support operators like `+`, `-`, `*` and `/`.

Let's start by defining the wrapper:

```
wrapBigInteger X:
  X
  _+_ Y: X.add[Y]
  _-_ Y: X.subtract[Y]
  _*_ Y: X.multiply[Y]
  _/_ Y: X.divide[Y]
  $_ = X.toString
```

This wrapper will take an instance of `Host.class("java.math.BigInteger")` and extend it with the operators we desire.

Now we *register* this wrapper so that the JPiccola runtime system will *always* apply our wrapper when it sees a BigInteger:

```
registerWrapper "java.math.BigInteger" wrapBigInteger
```

Of course, ordinary numbers are still wrapped in the usual way, so we need a way to convert a plain integer to a wrapped BigInteger. One way to do this in Java is to use the BigInteger(String) constructor:

```
big N: class("java.math.BigInteger").new[$N]
```

No we redefine `fact()` so the result is always a BigInteger:

```
def fact N:
  assert N>=0
  if N==0
    then: big(1)
    else: big(N)*fact(N-1)
```

Finally we obtain the result we expect:

```
println "fact(20) = " + fact(20)
```

```
fact(20) = 2432902008176640000
```

Exercise 40 *Explore a* BigInteger *before and after registering the wrapper. NB: use* `explore class("java.math.BigInteger").new("1")`

Exercise 41 *Define the remaining Number operators for wrapped* BigIntegers*. Define a* `PiUnit` *test suite that exercises these operators.*

Exercise 42 *Define a linear-time Fibonacci function that uses wrapped* BigIntegers*.*

Chapter 3

Piccola Standard Library

When JPiccola is started, the following standard library files are loaded:

- prelude.picl: sets up the initial `root` with standard services and forms. prelude.picl in turn loads the following two files.
- collections.picl: provides Lists, Maps and Sets
- piunit.picl: provides the testing framework

You can find these files in the directory source/piccola/lib of the JPiccola distribution, or in the directory lib of JPiccola3.7a.jar. (Unpack it with the command jar xvf JPiccola3.7a.jar .)

In this chapter we first document the services provided by the standard Piccola prelude, and then we document the built-in types used by these services.

3.1 The Standard `root`

After loading the prelude, the resulting initial `root` looks like this:

```
About = ...
Basic = ...
Collections = ...
Debug = ...
DefaultOp = ...
Host = ...
Kernel = ...
PiUnit = ...
asException = [service],
# various standard services ...
```

From within the JPiccola console, you may explore the current `root` by running `explore root`. This `root`, however, has been extended by services needed to run the console itself. In order to explore the standard root, you should evaluate the following script:

```
(loadCore "ide.picl" root).explore root
```

This loads the ide services into a new form, and projects and evaluates just the `explore` service on the standard `root`. The resulting explorer is illustrated in Figure 3.1.

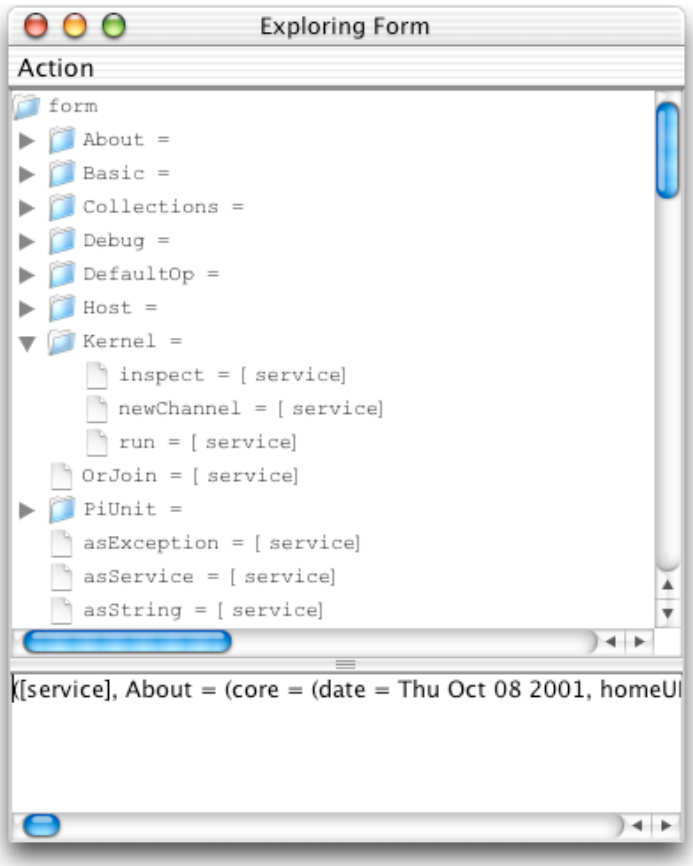

Figure 3.1: Exploring `root`

3.1.1 About

`About` documents the current version of Piccola.

Service	Type	Description
`core`	`version=`*String*`,` `date=`*String*`,` `homeURL=`*String*	Library version and build date
`vm`	`sys=`*String*`,` `version=`*String*`,` `date=`*String*	System (e.g. "java" or "squeak"), vm version and build date

3.1.2 Basic

`Basic` contains definitions of standard services that are also found in `root`. As a consequence, you may override these `root` services, while still providing access to the original services in `Basic`. For example:

```
println X:
  Basic.println "jpiccola " + About.vm.version + ">>> " + X
println "hello world"
```

yields:

```
jpiccola 3.6b>>> hello world
```

This resets println to the standard definition:

```
println = Basic.println
```

The interfaces of the types *Boolean*, *String*, *Number*, *Exception*, *Label*, *Blackboard*, and *Var* are defined below in Section 3.2.

A *Service* is an arbitrary Piccola service. An *URL* is a wrapped java.net.URL (see the Java API).

Service	Type	Description
`asException`	*String* → *Exception*	Create an exception from the argument. See Section 3.2.5.
`asString`	*Any* → *String*	Convert the argument `arg` into a String by invoking `$arg`.
`asService`	*Service* → `do:`*Service*	If necessary, convert `arg` so it can be invoked as `arg.do()`. (Used by `foreach` and `run`.)
`assert`	*Boolean* → ()	Raise an exception if `arg` is false.
`equals`	*Any* → *Any* → *Boolean*	Compare the arguments. Services are always considered to be different.
`eval`	*String* → *Any* → *Any*	Parses `arg1` and evaluates it with `arg2` as `root`.
`execNative`	*Any* → *Any*	Evaluates `(java:(), X).java` in JPiccola or `(squeak:(), X).squeak` in SPiccola.
`false`	*Boolean*	The boolean false value.
`for`	`(from:`() → *Number*`, to:`() → *Number*`, do:`*Number* → *Any*`)` → ()	Perform `arg1.do(`n`)` for n ranging from `arg1.from()` to `arg1.to()`. Defaults for `from()` and `to()` are `1`, and for `do()` is `()`.
`foreach`	`(hasNext:`() → *Boolean*`, next:`() → *Any*`)` → *Service* → ()	Perform `arg2(arg1.next())` while `arg1.hasNext()`.
`forEachLabel`	(`form` = ..., `do` : ...) → *Any*	Iterates over the labels of `arg.form` and runs `arg.do(L)` for each label `L`.
`hash`	*Any* → *Number*	Returns a hash value for `arg`.
`homeURL`	*String*	The URL of the current script. If the script has no URL, then this is the empty form.
`if`	*Boolean* → *Any* → *Any*	Returns `arg2.then()` or `arg2.else()` if `arg1` is `true` or `false`. The `then` and `else` bindings are optional.
`isEmpty`	*Any* → *Boolean*	Returns `true` if `arg` is the empty form
`isExternal`	*Any* → *Boolean*	Returns `true` if `arg` is a wrapped host component.
`isPeer`	*Any* → *Boolean*	Returns `true` if `arg` is a peer form.
`isService`	*Any* → *Boolean*	Returns `true` if `arg` is (or has) a service.
`label`	*Any* → *Label*	Returns a first class label contained in F or the empty form

`load`	*String* → *Any* → *Any*	Evaluates the piccola file identified by the URL `arg1` in the `root` context `arg2`. Returns the resulting context, with `About.homeURL` set to the URL of the loaded script.
`loadCore`	*String* → *Any* → *Any*	Load a file relative to the prelude.
`loadRelative`	*String* → *Any* → *Any*	Load a file relative to the current script's URL.
`loop`	(`while` : ..., `do` : ...) → ()	Evaluate `arg.do()` while `arg.while()` returns `true`.
`newBlackboard`	() → *Blackboard*	Creates a blackboard with `write`, `remove`, `read` services. See Section 3.2.13.
`newCounter`	*Number* → *Counter*	Returns a counter. See Section 3.2.12.
`newLabel`	*String* → *Label*	Returns a first class label with the name `arg`. See Section 3.2.7.
`newURL`	*String* → *URL*	Create a wrapped java.net.URL from `arg`. The string must conform to the URL specification.
`newVar`	() → *Var*	Create a new variable. See Section 3.2.11.
`print`	*Any* → ()	Prints the string `$arg`.
`println`	*String* → ()	Prints the string `$arg` followed by a newline.
`protect`	*Any* → *Any*	Wrap an extended Host form. See [Kne03], p. 26.
`raise`	*Any* → ()	Raise an exception.
`stop`	() → ()	Stop the current thread
`true`	*Boolean*	The boolean true value.
`try`	(`do` : ..., `catch` : ...) → ()	Evaluate `arg.do()` with the exception handler `arg.catch()`.
`typeOf`	*String* → *String*	*Experimental* type inference service. See [Kne03].

3.1.3 Collections

List, *Set* and *Map* are normally created with the global collection operators `[_]`, `{_}` and `{{_}}` (Section 3.1.5), and are described in Section 3.2.15.

Service	Type	Description
`newList`	() → *List*	Returns a wrapped Host list.
`newMap`	() → *Map*	Returns a wrapped Host hash table.
`newSet`	() → *Set*	Returns a wrapped Host set.

3.1.4 Debug

These are some miscellaneous services useful for debugging. *StackTrace* is documented in Section 3.2.16.

Service	Type	Description
`getStackTrace`	(`caller` = ...) → *StackTrace*	If `arg` is `dynamic`, returns the current stack trace. See Section 3.2.16.
`printLabels`	*Any* → ()	Prints the labels of an arbitrary form.
`printStackTrace`	*StackTrace* → ()	Prints a stack trace.
`wrapString`	*Any* → *String*	Returns a built-in debug representation of the argument.

To print the stack trace at any point in a Piccola script, you could invoke:

```
Debug.printStackTrace(Debug.getStackTrace(dynamic))
```

3.1.5 DefaultOp

The form `DefaultOp` defines global infix, prefix and collection operators. Global operators may be locally overridden, for example, `a+b` is interpreted as `a._+_(b)`, if `a` provides `_+_`. If not, then it is interpreted as `DefaultOp._+_default(a)(b)`. If the global `+` is not defined, an exception is raised.

The default operators `[_]`, `{_}` and `{{_}}` are defined in collections.picl. *List*, *Set* and *Map* are described in Section 3.2.

Service	Type	Description
`$_default`	*Any* → *String*	Converts the argument to a String.
`_==_default`	*Any* → *Any* → *Boolean*	The arguments are equal.
`_!=_default`	*Any* → *Any* → *Boolean*	The arguments are not equal.
`_->_default`	*Any* → *Any* → (`key` = ..., `value` = ...)	Returns a form binding `arg1` to `key` and `arg2` to `value`.
`[_]`	() → *List*	Returns a wrapped Host list.
`{_}`	() → *Set*	Returns a wrapped Host set.
`{{_}}`	() → *Map*	Returns a wrapped Host hash table.

New global infix, prefix and collection operators can be defined by updating `DefaultOp`. For example,

```
DefaultOp.!_default x: println x
```

defines `!` to be a global prefix operator that prints its argument. Now:

```
!"howdy!"
```

yields:

```
howdy!
```

3.1.6 Host

Here various (Java) host-specific services are defined.

Service	Type	Description
`class`	*String* → *Class*	Returns a form representing a Java class. See Section 3.2.9.
`classOf`	*WrappedJavaObject* → *Class*	Returns a form representing a Java class. See Section 3.2.9.
`exit`	() → ()	Stop the Java vm.
`false`	*Boolean*	Boolean false.
`getProperty`	*String* → *String*	Looks up a Piccola system property. (See javadoc of ch.unibe.piccola.Host.getProperty)
`getResource`	*String* → *URL*	Returns a a wrapped java.net.URL for a filename.
`meta`	*Any* → *Meta*	Returns a meta-representation of a form. See Section 3.2.8.
`newLocalClassLoader`	*String* → ()	Sets the class loader from a local file.
`newString`	() → *StringBuffer*	Returns a wrapped Java StringBuffer.
`newURLClassLoader`	*String* → ()	Sets the class loader from a URL.
`registerWrapper`	*String* → *Service* → ()	Used to register the service `arg2` as a wrapper for the class `arg1`.
`setAutoAbort`	*Boolean* → ()	If set to `true`, Host terminates automatically when all agents are blocked or terminated; otherwise terminates on `Host.exit()`.
`setCallDynamic`	*Any* → ()	Sets the dynamic context for callbacks.
`true`	*Boolean*	Boolean true.

3.1.7 Kernel

These services are available in `root`, and are provided by the Piccola kernel.

Service	Type	Description
`inspect`	*Any* → `(isLabel:..., isService:..., isEmpty:...)` → ()	If `arg1` contains a label `L`, calls `arg2.isLabel(L)`; else, if it contains only a service, calls `arg2.isService()`; else (if it is empty), calls `arg2.isEmpty()`;
`newChannel`	() → *Channel*	Creates a new Channel. See Section 3.2.10.
`run`	`do` : ... → ()	Runs `arg.do()` in a concurrent agent.

3.1.8 PiUnit

This form provides a simple unit testing framework. Tests are normally provided in scripts by a `test` service. For examples, see the core tests defined for the standard library (folder test relative to prelude.picl).

Service	Type	Description
`assert`	*Boolean* → ()	Raises an exception if `arg` is not true.
`assertEquals`	*Boolean* → ()	Raises an exception if `arg.expect` does not equal `arg.get`.
`assertFails`	`do` : ... → ()	Raises an exception if `arg.do()` does *not* raise an exception.
`assertFalse`	*Boolean* → ()	Raises an exception if `arg` is not false.
`fails`	() → ()	Raises an assertion exception.
`loadCoreTest`	*String* → *Any* → *Any*	Loads a core test.
`loadTest`	*String* → *Any* → *Any*	Loads a test relative to current script.
`newTest`	(`name`=*String*, `test:...`) → *Test*	Builds a test case with name `arg.name` and body `arg.test()`. See Section 3.2.6.

Note that both `Basic` and `PiUnit` provide `assert` services. They essentially behave the same way, except that `PiUnit.assert` generates a different message, and it returns a *StackTrace* relative to the actual location of the failed test.

When writing scripts, a good strategy is to always return a binding `test` to a suite of tests that exercise the services defined in the script. Such a script can either be evaluated, *i.e.*, causing its `main` service to be run, or loaded to run its `test` service.

Alternatively, you may put all the tests in a separate file. In this case, each test script should provide a `main` service that will run the tests. That way, when the test case is run as a script, `main` will be invoked and the tests will be run. On the other hand, when the script is loaded from another script, the `test` service can be composed with other tests. This strategy is used to test the standard libraries. There is a test file for each logical group of services (*i.e.*, testBasic.picl, testBool.picl, and so on), and a master script, testAll.picl, which loads them all, composes them into a single test suite, and runs them all. (See the test directory within the lib directory containing prelude.picl.)

3.1.9 Builtin Host

The following services are provided by Piccola when it starts up, and are renamed or wrapped by the prelude.picl.

Service	Type	Description
FALSE	*Boolean*	false
TRUE	*Boolean*	true
exit	() → ()	Host.exit
findResource	*String* → *URL*	Host.getResource
getProperty	*String* → *String*	Host.getProperty
meta	*Any* → *Meta*	Host.meta
newChannel	() → *Channel*	Kernel.newChannel
newClass	*String* → *Class*	Host.class
newVariable	*Any* → *Variable*	Used by Host.newVar
print	*String* → ()	Basic.print
println	*String* → ()	Basic.println
registerWrapper	*String* → *Service* → ()	Host.registerWrapper
serviceFromString	*String* → *Any* → *Service*	Used to define Basic.eval
serviceFromURL	*URL* → *Any* → *Service*	Used to define Basic.load
setAutoAbort	*Boolean* → ()	Host.setAutoAbort
setCallDynamic	*Any* → ()	Host.setCallDynamic
setExceptionHandler	*Service* → ()	Used to set Basic.raise as the exception handler.
toString	() → *String*	Returns host version info (not used)
typeFromString	*String* → *String*	Basic.typeOf

Recall that an *URL* is a wrapped java.net.URL

3.2 Builtin Types

3.2.1 Boolean

The JPiccola Boolean wraps, but does not export all the services of a Java Boolean.

Service	Type	Description
and	*Boolean* → *Boolean*	Boolean *and*.
not	() → *Boolean*	Boolean negation.
or	*Boolean* → *Boolean*	Boolean *or*.
select	(true=*Any*, false=*Any*)→ *Any*	Returns arg.true or arg.false.
!	() → *Boolean*	Boolean negation.
$	() → *String*	Returns “true” or “false”.
&	*Boolean* → *Boolean*	Boolean *and*.
&&	*Boolean* → *Boolean*	Lazy Boolean *and*.
|	*Boolean* → *Boolean*	Boolean *or*.
||	*Boolean* → *Boolean*	Lazy Boolean *or*.

3.2.2 String

A JPiccola *String* supports all the services of a java.lang.String, in addition to the following services:

Service	Type	Description
`charAt`	*Number* → *Char*	Returns the (wrapped) Java character at the given position.
`indexOf`	*Char* → *Number*	Returns the index of the given character within the string.
`isEmpty`	() → *Boolean*	Returns whether size is zero.
`size`	*Number*	The length of the string.
`substring`	(`from=`*Number*`, to=`*Number*) → *String*	Returns the substring indexed by the given range.
`$`	() → *String*	Returns the string representation.
`+`	*String* → *String*	String concatenation.
`<=`	*String* → *Boolean*	Alphabetical ≤.
`<`	*String* → *Boolean*	Alphabetical <.
`>=`	*String* → *Boolean*	Alphabetical ≥.
`>`	*String* → *Boolean*	Alphabetical >.

3.2.3 StringBuffer

A JPiccola *StringBuffer* supports all the services of a java.lang.StringBuffer, in addition to the following services:

Service	Type	Description
`append`	*String* → *StringBuffer*	Append `arg`.
`$`	() → *String*	Returns the String representation.
`+`	*String* → *String*	String concatenation.

3.2.4 Number

A JPiccola *Number* supports all the services of a java.lang.Number, in addition to the following services:

Service	Type	Description
`abs`	() → *Number*	Absolute value.
`asByte`	() → *WrappedJavaObject*	java.lang.Number.byteValue
`asDouble`	() → *WrappedJavaObject*	java.lang.Number.doubleValue
`asFloat`	() → *WrappedJavaObject*	java.lang.Number.floatValue
`asInteger`	() → *WrappedJavaObject*	java.lang.Number.intValue
`asLong`	() → *WrappedJavaObject*	java.lang.Number.longValue
`asShort`	() → *WrappedJavaObject*	java.lang.Number.shortValue
`equals`	() → *Boolean*	java.lang.Number.equals
`smaller`	*Number* → *Boolean*	<
`trunc`	() → *Number*	Truncate the number.
`$`	() → *String*	String representation.
`-`	() → *Number*	Negation
`%`	*Number* → *Number*	Modulo.
`*`	*Number* → *Number*	Multiplication.
`+`	*Number* → *Number*	Addition
`-`	*Number* → *Number*	Subtraction.
`/`	*Number* → *Number*	Division.
`<=`	*Number* → *Boolean*	≤
`<`	*Number* → *Boolean*	<
`==`	*Number* → *Boolean*	Equals.
`>=`	*Number* → *Boolean*	≥
`>`	*Number* → *Boolean*	>

3.2.5 Exception

This is not a Java Exception, but a JPiccola representation of an exception. In any case, when a Java Exception is raised by a wrapped Java object, it will be caught by the JPiccola runtime and wrapped as a JPiccola *Exception*.

Service	Type	Description
`getStackTrace`	() → *StackTrace*	Return the StackTrace at the point the exception was raised. See Section 3.2.16.
`msg`	*String*	Textual representation of the exception.
`defaultAction`	() → ()	Default action (usually, print the msg)

3.2.6 Test

A single test case is built using `PiUnit.newTest` (see Section 3.1.8). Test suites are built by composing tests with the `+` operator. A test is run by invokling its `test` service, with an optional `verbose=true` flag, which reports the name of each individual test case being run.

Service	Type	Description
`test`	`verbose=`*Boolean* → ()	Run the test.
`+`	*Test* → *Test*	Create a composite test suite.

3.2.7 Label

First-class labels are returned by the services `Basic.label` and `Basic.newLabel`. (The former takes *Any* form as its argument and returns a first-class label representing the "next" label bound in that form; the latter constructs a first-class label whose name is the argument *String*.)

Service	Type	Description
`bind`	*Any* → *Any*	Returns the form ***`label`*`=arg`**, where ***`label`*** is the name of the label.
`exists`	*Any* → *Boolean*	Returns whether ***`label`*** is bound in `arg`.
`hide`	*Any* → *Any*	Removes any binding for ***`label`*** present in `arg`.
`name`	() → *String*	Returns ***`label`***.
`project`	*Any* → *Any*	Returns `arg.`***`label`***; else raises an exception.
`restrict`	*Any* → *Any*	Same as hide.
`$`	*String*	Returns `"Label(`***`label`***`)"`
`==`	*Any* → *Boolean*	Equality test

3.2.8 Meta

`meta f` returns a meta-representation of a form `f` that provides some convenient reflective services.

Service	Type	Description
`bind`	*String* → *Any*	Returns `arg=f`
`equals`	*Any* → *Boolean*	`meta f` equals `meta arg`?
`exists`	*String* → *Boolean*	Is `arg` bound in `f`?
`extend`	*Any* → *Any*	Extend `f` by `arg`
`getBindings`	() → *Map*	Returns map of bindings in `f`.
`hashCode`	() → *Number*	Same as `Basic.hash(f)`
`hide`	*String* → *Any*	Hide label `arg` in `f`
`isEmpty`	() → *Boolean*	Is form the empty form?
`isPeer`	() → *Boolean*	Is form a peer form?
`isService`	() → *Boolean*	Does `f` have a service?
`labels`	() → (`hasNext:`() → *Boolean*, `next:`() → *Any*)	Returns an iterator over labels (cf. `Basic.foreach`).
`project`	*String* → *Any*	Lookup `f.arg`
`runtimeEquals`	*Any* → *Boolean*	Compares `f` and `arg`. Called by global default `==`.
`runtimeToString`	() → *String*	Converts `f` to a *String* using global `$`
`toString`	() → *String*	String representation of `f`

3.2.9 Class

The JPiccola service `Host.class` returns a form that wraps services of the named Java class.

Service	Type	Description
`getClass`	() → *WrappedJavaObject*	Returns a wrapped java.lang.Class
`new`	() → *WrappedJavaObject*	Returns a wrapped instance of the class.
`newArray`	*Number* → *Array*	Returns an array of wrapped instances of the class.

3.2.10 Channel

Channels are created using `Kernel.newChannel()`.

Service	Type	Description
`receive`	() → *Any*	Blocking receive; consumes and returns a value from the channel.
`send`	*Any* → ()	Non-blocking send; puts a value onto the channel.

3.2.11 Var

Variables are created with `Basic.newVar(`*Any*`)`. The optional argument is the initial value of the variable (by default, the empty form).

Service	Type	Description
`get`	() → *Any*	Return the current value of the variable.
`set`	*Any* → *Any*	Update the variable and return the new value.
`*`	() → *Any*	Same as get.
`<-`	*Any* → *Any*	Same as set.

Note that a *Var* is simple an extended *Variable* (provided by the `newVariable` service of the built-in Host), that adds `*` and `<-` as syntactic sugar for `get` and `set`.

3.2.12 Counter

A Counter is created with `Basic.newCounter(`*Number*`)`. A Counter extends a variable with the following services:

Service	Type	Description
`dec`	() → *Number*	Decrement the counter and return the updated value.
`inc`	() → *Number*	Increment the counter and return the updated value.

Note that initializing a *Counter* with a non-*Number* will result in an exception being raised when the counter is incremented or decremented.

3.2.13 Blackboard

A Blackboard wraps a Channel, renames `send` and `receive` to `write` and `remove`, and provides a non-destructive `read` service. A Blackboard can be created with `Basic.newBlackboard`.

Service	Type	Description
`read`	() → *Any*	Non-destructively reads and returns a value from the blackboard.
`remove`	() → *Any*	Removes and returns a value from the blackboard.
`write`	*Any* → ()	Non-blocking send; puts a value onto the blackboard.

3.2.14 Array

Array forms are peer forms of Java array objects, and are returned by any wrapped Java object whose methods return Java Arrays.

Service	Type	Description
`at`	*Number* → *Any*	Element lookup
`forEach`	*Service* → ()	Iterate over elements
`size`	() → *Number*	Size of array

3.2.15 Collections

Piccola supports Lists, Sets and maps, which are created, respectively, using the global default operators `[_]`, `{_}` and `{{_}}`.

Every type of collection supports the following services;

Service	Type	Description
`add`	*Any* → *Collection*	Append `arg` to the end of the collection.
`contains`	*Any* → *Boolean*	The collection contains `arg`.
`forEach`	*Service* → ()	Invokes `arg(`*item*`)` (alternatively, `arg.do(`*item*`)`), for each item.
`remove`	*Any* → *Collection*	Removes the first occurrence of `arg` from the collection.
`size`	() → *Number*	The current length of the collection.
`map`	*Service* → *Collection*	Return the collection the results from applying `arg` (resp. `arg.do`) to each element of the collection.

`reduce`	*Service* → *Any* → *Any*	Produce the product of the collection, using `arg1` as the operator, and `arg2` as the initial value.
`+`	*Any* → *Collection*	Return a copy of the collection with element `arg` appended to the end.
`++`	*Collection* → *Collection*	Return a copy of the collection with collection `arg` concatenated to the end.
`-`	*Any* → *Collection*	Return a copy of the collection with element `arg` removed.
`--`	*Collection* → *Collection*	Return a copy of the collection with every occurrence of `arg` removed.
`?`	*Any* → *Boolean*	The collection contains `arg`.
`$`	() → *String*	Returns the string representation of the collection

List Lists additionally support the following services:

Service	Type	Description
`at`	*Number* → *Any*	Returns the item at position `arg` (NB: the first item is at position1, not 0).
`set`	`(index=`*Number*`, elem=`*Any*`)` → *Any*	Updates the value at position `arg.index` (must be a valid index); returns the old value.
`removeAll`	*Any* → *List*	Removes every occurrence of `arg` from the list.

Set Sets additionally support the following services:

Service	Type	Description
`subset`	*Set* → *Boolean*	The set is a subset of `arg`.
`superset`	*Set* → *Boolean*	`arg` is a subset of the set.
`<=`	*Set* → *Boolean*	The set is a subset of `arg`.
`>=`	*Set* → *Boolean*	`arg` is a subset of the set.

Map Maps additionally support the following services:

Service	Type	Description
`at`	*Any* → *Any*	Lookup key `arg`.
`get`	*Any* → *Any*	Lookup key `arg`.
`containsKey`	*Any* → *Boolean*	`arg` occurs as a key.
`containsValue`	*Any* → *Boolean*	`arg` occurs as a values.
`isEmpty`	() → *Boolean*	No keys are bound.
`keys`	() → *Set*	Returns the set of keys.

3.2.16 StackTrace

A StackTrace is returned by `getStackTrace(dynamic)`, or by `e.getStackTrace`, where `e` is a raised exception.

Service	Type	Description
`bottom`	*Boolean*	We are at the bottom of the stack.
`src`	*String*	The current source code location.
`top`	() → *String*	Returns the current source code location.
`pop`	() → *StackTrace*	Returns the caller's StackTrace

Chapter 4

Piccola Language

The following description of the Piccola syntax has been adapted from Franz Achermann's PhD thesis [Ach02]. Please consult the thesis for details of the formal semantics.

4.1 The Language

We now define the syntax of the Piccola language. The language does not contain syntactical primitives for communication along channels, for spawning off new agents, and for hiding labels. These features are made available as predefined services in the initial root context.

4.1.1 Abstract Syntax

The abstract syntax of Piccola is given in Table 4.1. The grammar is a set of productions that describe how form expressions are constructed. Terminal symbols of the grammar are the Piccola keywords **def** and **root**, the symbols backslash (\), colon (`:`), dot (`.`), round parentheses (`()`), comma (`,`), equal (`=`), and the quote sign (`'`). Nonterminal symbols are shown in *Italic*. Optional parts for an alternative are written in square brackets [...].

The most important class of terms are Piccola form expressions. These expressions evaluate to a form. Constant *literals* are numbers and strings. Strings are enclosed in double quotes (`"`) or in triple douple quotes (`""" ... """`). The former interpret escaped characters, whereas the latter do not.

Normal *identifiers* start with an alphanumeric character and are followed by a sequence of numeric, alphanumeric and underscore characters. Special identifiers start with an underscore or an operator character, are followed by a sequence of alphanumeric, numeric and operator characters, and end with an underscore possibly followed by a sequence of alphanumeric characters.

Special identifiers denote the user-defined operators labels. The underscore is a placeholder for arguments of infix and prefix operators. For instance, the identifier for the `+` infix operator is `_+_`, its default label `_+_default`. *User-defined operators* are sequences of the characters: `* / + - = < > ! % : ; ~ ^ $ | ? &` and `@`. Alternatively, an operator can be an identifier written in backquotes, like `` `mod` ``. Collections are enclosed by tokens $op^{\{}$ and $op^{\}}$ that match. These tokens are sequences of `{` or `[` and `}` or `]`, respectively. They match if their individual characters match in reverse order. For instance, `[{{` matches with `}}]`.

4.1.2 Precedence Rules

The precedence rules for Piccola are given in Table 4.2. Each syntactical category has a precedence and an associativity. For instance the application "`a b`" has precedence 3 and is left-associative. Subterms may only have higher precedence. The precedence of infix expression is given by the first character of the infix operator. There are four groups of precedence:

Form ::=	
`root`	*current namespace*
identifier	*label*
literal	*constant literal*
\[*Param*] : *Form*	*anonymous service*
Form . *identifier*	*projection*
Form Form	*application*
Form op Form	*infix application*
op Form	*prefix application*
Form , *Form*	*extension*
op{ [*FormList*] *op*}	*collection*
([*Form*])	*parentheses*
`root` = *Form* [, *Form*]	*sandbox*
[`def`] *Label* [*Param*] : *Form* [, *Form*]	*service binding*
[`def`] *Label* = *Form* [, *Form*]	*binding*
' *Form* [, *Form*]	*quote*
FormList ::=	
[*FormList* ,] *Form*	*collection composition*
Param ::=	
identifier [*Param*]	
([*identifier*]) [*Param*]	
Label ::=	
identifier	*simple label*
Label . *identifier*	*nested label*

Table 4.1: Piccola Language Syntax

Precedence	Category	Concrete Syntax
9R	prefix	*op Form*
8L	projection	*Form* . *identifier*
	tight invocation	*Form*(*Form*)
7L	arithmetic high	*Form op Form*
6L	arithmetic low	*Form op Form*
5L	comparison	*Form op Form*
4L	other op	*Form op Form*
3L	invocation	*Form Form*
2R	service	\[*Param*] : *Form*
	binding	[`def`] *Label* = *Form*
	service binding	[`def`] *Label* [*Param*] : *Form*
	sandbox	`root` = *Form*
	quote	' *Form*
1L	collection composition	*FormList* , *Form*
1R	binding sequence	[`def`] *Label* = *Form* , *Form*
	service binding sequence	[`def`] *Label* [*Param*] : *Form* , *Form*
	sandbox sequence	`root` = *Form* , *Form*
	quote sequence	' *Form* , *Form*
	extension	*Form* , *Form*

Table 4.2: Precedence Rules

	Group	First character
7L	Arithmetic high	`* /`
6L	Arithmetic low	`+ -`
5L	Comparison	`= < > !`
4L	Other	`% : ; ~ ^ $ | ? & @` *identifier*

As an example, the expression "`a b +> c`" is parsed as "`a (b +> c)`" because arithmetic low (defined by the `+` character) has precedence 6 which is higher than precedence of invocation which in turn is 3. If the precedence is left or right associative, then the left or right subterm may in addition have the same precedence. The expression "`a b c`" is parsed as "`(a b) c`" because invocation is left associative.

Note that infix operators have higher precedence than invocation. Therefore the expression "`a + b`" is interpreted as an infix arithmetic expression and not as "`a (+ b)`".

There are two different precedences for invocation: tight invocation (8L) and normal invocation (3L). For tight invocation, the argument must be enclosed in parentheses or collection brackets and must immediately follow the functor. No whitespace may occur in between. For instance, the expression "`a b(c)`" is parsed as "`a (b c)`", whereas "`a b (c)`" is parsed as "`(a b) c`". The motivation to distinguish normal and tight invocation is driven by the desire to write code as

```
a().b(c = x).d    instead of  ((a()).b(c = x)).d and
a b c             instead of  a(b)(c).
```

The precedence rules are strict. This means that they not only rule out ambiguities but they also forbid certain constructs and force the programmer to use parentheses or indentation. For instance the expression "`a b=()`" would be syntactically valid if parsed as "`a (b=())`". However, such a parsing is not permitted since the expression "`b=()`" has precedence 2 and cannot be a subterm of invocation with precedence 3. Such an expression must be corrected using additional parentheses or indentation.

The associativity induced by a comma is different when the comma appears as top-level operator in an expression sequence or within collection brackets.

Collection. A comma appearing top-level inside collection brackets has precedence 1L and denotes an expression *FormList, Form*. For instance the expression "`[a, b = x, c]`" is parsed as "`[(a, (b = x)), c]`". This means that the collection will contain three elements: the value of "`a`", of "`b = x`", and that of "`c`".

Note that by using parentheses the meaning of the comma changes: The collection "`[a, (b = x, c)]`" contains two elements, namely "`a`" and the term "`b = x, c`".

Sequence. A comma appearing in a sequence of form expressions has precedence 1R. If the left hand side of a comma is a binding, a service binding, a sandbox or a quote expression, then the right-hand side is the *scope* of the left hand side. In these cases the value of the left-hand side extends the current root context for the scope. For instance the expression

```
a = 1, 'b, c
```

is parsed as "`(a = 1, ('b, c))`", i.e., the expression "`'b, c`" is in the scope of the binding "`a = 1`" and the expression "`c`" is in the scope of "`b`".

4.1.3 Indentation

Piccola supports indentation and newlines instead of parentheses and comma to group form expressions. For example the term "`x = f(a = (), b = a), y`" is normally written as

```
x = f
  a = ()
  b = a
y
```

When a line starts at a higher or lower indentation than the previous line, an opening parenthesis (indent) or a closing parenthesis (dedent) is inserted, respectively. If the new line starts on the same indentation level, a comma is inserted. Inserted dedents may not mix matching parentheses, brackets or other indent-dedent pairs. Therefore, one or multiple dedent tokens are inserted before any closing parentheses, bracket or dedent, if a corresponding indent was inserted but not closed between the matching pairs. The precise rules are as follows. Assume line n has indentation level d. The following line has indentation level d'.

Indent. If $d < d'$ an indent with indentation level d' is inserted unless line n ends with an opening parentheses or bracket or the following line starts with a dot.

Comma. If $d = d'$, a comma is inserted unless line n ends with a comma, an operator, an opening parentheses or bracket, or the following line starts with a dot.

Dedent. If $d > d'$ and the following line does not start with a closing parentheses or bracket then closing dedents to match previously inserted indents are inserted until there is a remaining unmatched opening parentheses, bracket, or an indent with a lower indentation level. A comma is inserted unless the following line starts with a dot.

Closing. Dedents are inserted before any closing parentheses or bracket if there are unmatched indents inserted after the matching parentheses or bracket that gets closed. If dedents are inserted, the next line must not have an indentation level higher than the last inserted dedent.

End. At the end of an input, as many dedents are inserted as there are remaining unmatched indents inserted.

The precedence of invocations with an indented argument have normal precedence (8L). For example

```
a b
  c
```

is parsed as "`(a b) c`".

Indentation restricts the programmer's freedom to insert newlines at will. For instance a newline cannot occur after an equal sign unless the value of the binding is indented or put into parentheses. For instance

```
a =
b
c
```

is tokenized as "`a = , b , c`" which is syntactically wrong.

Note that no indents are inserted if the previous line ends with an opening bracket. For example the code

```
a = [
  1
  2]
```

is read as "`a = [1, 2]`". This collection has two elements, whereas "`a = [(1, 2)]`" denotes a collection with one element, namely 1 extended with 2.

4.2 Abbreviations

Many of the features of the Piccola language are syntactic sugar. In Table 4.3 the expansion for these features are given. We use T to range over form expressions, P over parameter expressions and L over label expressions. The abbreviations define, amongst other simplifications, the semantics for user-defined operators and collections.

$$
\begin{array}{rcll}
[\ \texttt{def}\ ]\ L\ [\ P\ ]\ \texttt{:}\ T & \equiv & [\ \texttt{def}\ ]\ L\ \texttt{=}\ \texttt{\textbackslash}[\ P\ ]\ \texttt{:}\ T & \text{(abb-sd)} \\
\texttt{\textbackslash}I\ P\ \texttt{:}\ T & \equiv & \texttt{\textbackslash}I\ \texttt{:}\ \texttt{\textbackslash}P\ \texttt{:}\ T & \text{(abb-curry1)} \\
\texttt{\textbackslash(}[\ I\ ]\texttt{)}\ P\ \texttt{:}\ T & \equiv & \texttt{\textbackslash(}[\ I\ ]\texttt{)}\ \texttt{:}\ \texttt{\textbackslash}P\ \texttt{:}\ T & \text{(abb-curry2)} \\
\texttt{\textbackslash}I\ \texttt{:}\ T & \equiv & \texttt{\textbackslash(}I\texttt{)}\ \texttt{:}\ T & \text{(abb-paren)} \\
\texttt{\textbackslash:}\ T & \equiv & \texttt{\textbackslash()}\ \texttt{:}\ T & \text{(abb-void)} \\
[\ \texttt{def}\ ]\ L\ \texttt{.}\ I\ \texttt{=}\ T & \equiv & L\ \texttt{=}\ \texttt{(}L\ \texttt{,}\ [\ \texttt{def}\ ]\ I\ \texttt{=}\ T\texttt{)} & \text{(abb-nest)} \\
[\ \texttt{def}\ ]I\ \texttt{=}\ T_1\texttt{,}\ T_2 & \equiv & \texttt{'}\ \texttt{(}[\ \texttt{def}\ ]I\ \texttt{=}\ T_1\texttt{)},\ \texttt{(}I\ \texttt{=}\ I\texttt{)},\ T_2 & \text{(abb-assign)} \\
\texttt{'}\ T_1\ [\ \texttt{,}\ T_2\ ] & \equiv & \texttt{root}\ \texttt{=}\ \texttt{(root}\ \texttt{,}\ T_1\texttt{)}\ [\ \texttt{,}\ T_2\ ] & \text{(abb-quote)} \\
\texttt{root}\ \texttt{=}\ T & \equiv & \texttt{root}\ \texttt{=}\ T\texttt{,}\ \texttt{()} & \text{(abb-sandbox)} \\
T_1\ op\ T_2 & \equiv & \texttt{('(}x\ \texttt{=}\ T_1\texttt{)},\ op^{infix}\ \texttt{(}y\texttt{):}\ \texttt{(}op^{defaultInfix}\ x\ y\texttt{)},\ x\texttt{)}.op^{infix}\ T_2 & \text{(abb-infix)} \\
op\ T & \equiv & \texttt{('(}x\ \texttt{=}\ T\texttt{)},\ op^{prefix}\ \texttt{():}\ \texttt{(}op^{defaultPrefix}\ x\texttt{)},\ x\texttt{)}.op^{prefix}\ \texttt{()} & \text{(abb-prefix)} \\
op^{\{}\ op^{\}} & \equiv & op^{\{\}}\ \texttt{()} & \text{(coll-empty)} \\
op^{\{}\ [\ T_1,\ ]\ T_2\ op^{\}} & \equiv & \texttt{(}op^{\{}\ [\ T_1\ ]\ op^{\}}\ \texttt{)}\ \texttt{.}\ \texttt{add}\ T_2 & \text{(coll-add)}
\end{array}
$$

Table 4.3: Piccola Language Abbreviations

4.2.1 Services

The rule *abb-sd* allows the programmer to define services and bind them to an identifier in a single expression. For example

```
id x: x
```

is syntactic sugar for: `id = \x: x`. Observe that the name of the service `id` is not visible in the body of its binding. For recursive services we use the `def` keyword.

The rules *abb-curry1* and *abb-curry2* allow the programmer to write curried functions more user friendly. Instead of

```
\l: \r: l + r
```

we can write

```
\l r: l + r
```

The rules *abb-paren* and *abb-void* allow us to omit parentheses and formal parameters in parameter expressions. For instance in the example

```
if n < 2
  then: 1
  else: n * fac(n - 1)
```

the bindings for label `then` and `else` expand to:

```
if n < 2
  then = \(): 1
  else = \(): n * fac(n - 1)
```

4.2.2 Nested Bindings

The rule *abb-nest* specifies the semantics of bindings with a nested label. A nested binding "`a.b = c`" extends the form denoted by `a` with the binding "`b = c`". This is achieved by writing "`a = (a, b = c)`". This process can be repeated to unfold the complete structure of the nested label.

Observe that an expression `Label.x = T` is only valid when the current root context contains a binding for `Label`. This is due to the fact that the nested binding is translated to a binding where the right-hand side value is an extension of `Label` with a normal binding. For instance the following code is invalid

```
a = ()                  # a will be the empty form
a.b.c = ...             # wrong!
```

since the form `a.b` is not defined.

The rule *abb-nest* defines an inner fix-point when used with the `def` keyword. The term "`def f.b = c`" is equivalent to

```
f =
  f
  def b = c
```

whereas an outer fix-point would be:

```
def f =
  f                   # f used while being defined. Wrong!
  b = c
```

Such code is illegal since it would denote an infinite form.

4.2.3 Assignment

The rule *abb-assign* allows us to rewrite binding assignments with a nonempty scope. The value of "`l=a,b`" is a binding `l=a` extended with the value of "`b`" where "`b`" is evaluated in a context that contains `l=a`. This is achieved by rewriting "`l=a,b`" as:

```
'(l = a)
(l = l)
b
```

Note that the quoted expression evaluates "`a`" and extends the root context. The following example illustrates the difference between assignment and extension:

```
a =
  l = 1             # assignment
  println l         # prints 1
  (l = 2)           # extension, does not change root
  println l         # still prints 1
println a           # prints (l = 2)

1
1
(l = 2)
```

4.2.4 Quoted Expressions

We call expressions of the form "`root = ...`" *sandboxes* because they replace the `root` namespace of any expression that follows.

We extend the root context with the value of an expression "`a`" by:

```
root = (root, a)
...
```

Since such constructs occur very frequently we provide a special notation using quotes. By rule *abb-quote* the above code is abbreviated as:

```
'a
...
```

The quote construct is often used for local definitions. Its bindings are not exported.

The rule *abb-sandbox* defines the empty form as the scope for sandbox expressions without scope. The value of a quoted expressions is the empty form. For instance:

```
x = 'a
```

is syntactic sugar for "`x=(root =(root,a),())`". The value of the whole expression is the binding `x=()`.

These rules explain the behaviour of the following idiom of using two quotes in Piccola. For instance, "`''extern(), ...`" is an abbreviation for

```
root =
  root
  (root = (root, extern()), ())
...
```

This code evaluates `extern()` and extends the root context with the empty form. This means that the result of `extern()` is not used, the application is evaluated for its side effect only. We also use this idiom if the application is known to return the empty form to make the code more self-documenting.

4.2.5 User Defined Operators

The infix expression "`a + b`" is syntactic sugar for

```
(
  '(x = a)
  (_+_ y: DefaultOp._+_default x y)        # default +
  x
)._+_ b
```

The behaviour of this term is as follows. Assume the value of the expression `a` contains a binding `_+_=S`. In this case the projection on the last line denotes the service S and the expression is equivalent to

```
a._+_ b
```

The infix operator is dispatched on its left-hand expression.

Now, assume `a` has no binding for the label `_+_`. In this case the projection sees the service that is defined as a default operator and the infix-expression is equivalent to

```
DefaultOp._+_default a b
```

The order of evaluation is in both cases the same: First the left-hand expression is evaluated then the right-hand expression is evaluated and finally the service of the operator is applied. We use the label `DefaultOp` to contain global defaults for user-defined operators.

A similar expansion works for prefix operators. The term "`+a`" behaves as "`a.+_()`" when the form `a` contains the label `+_` and as "`DefaultOp.+_default a`" otherwise.

4.2.6 Collections

The semantics of user-defined collections in Piccola is specified by the rules *coll-empty* and *coll-add*. They work as follows. For each user-defined collection there is a global factory in `DefaultOp`. The code "`x = { }`" is syntactic sugar for

```
x = DefaultOp.{_}()
```

It should be noted that `DefaultOp.{_}` is not a collection itself, but a factory to create new (empty) collections. Individual elements are added to such a collection using its `add` service. The term "`{a, b, c}`" is syntactic sugar for

```
DefaultOp.{_}().add(a).add(b).add(c)
```

Observe that a collection, i.e., the form returned by the factory must contain a service `add` which in turn returns the collection with the added element.

Recall from Section 4.1.2 that the comma separates the elements of the collection. As such, the bindings do not behave like assignments. For instance

```
a = 1              # Assignment
x = [[
  a = 2            # inside collection
  b = a]]
```

is syntactic sugar for

```
a = 1
x = DefaultOp.[[_]]().add(a = 2).add(b = a)
```

The root context is the same for both elements that are added to the new collection. The collection x contains the bindings `a=2` and `b=1`.

Chapter 5

A Brief History of Piccola

Piccola was developed as a result of the desire to have a formal framework for understanding how software entities can be composed. In general, we assume that software systems are *open* and *evolving*, and in particular, *concurrent* and *distributed*. An early attempt to propose such a framework was outlined in "Viewing Objects as Patterns of Communicating Agents" [NP90]. *Abacus* was a first attempt to implement a Piccola-like language [Nie90].

A research agenda for developing a composition language was first described in the position paper "Requirements for a Composition Language" [NM95]. This led first of all to some experiments using Milner's π-calculus [MPW92], and its implementation in the Pict programming language [PT95], to model various sorts of software abstractions [NSL96, LSN96]. A key result of this work was the insight that extensible records, *i.e.*, *forms*, provided a better basis for modeling reusable software abstractions and wrappers than did tuples. This in turn led to the development of the $\pi\mathcal{L}$-calculus, which is presented in detail in the PhD dissertation of Markus Lumpe [Lum99]

Piccola itself was first proposed in the ESEC 97 Workshop on Foundations of Component-Based Systems [LSNA97]. The syntax of Piccola has evolved considerably since the first version, which we refer to as Piccola1. (The version described in the present document is Piccola3.) Piccola1 is defined on top of $\pi\mathcal{L}$, and is described in Lumpe's thesis and in the paper "A Formal Language for Composition" [LAN00].

The PhD thesis of Jean-Guy Schneider [Sch99] presents the Form calculus, a refinement of the $\pi\mathcal{L}$-calculus that further eases modeling of software abstractions, and illustrates how forms can be conveniently used to model a wide range of software composition mechanisms. The thesis also presents Piccola(F), a version of Piccola built on top of the Form calculus.

There are two papers presenting Piccola2 in some detail, "Applications = Components + Scripts – A tour of Piccola" [AN01] and "Piccola – a Small Composition Language" [ALSN01]. These papers also introduce the notion of a *compositional style*, which enables one to express an application as a composition of software components. The conceptual framework supported by Piccola is introduced in the paper "Components, Scripts and Glue" [SN99]. Compositional styles are explored in various papers [AKN00, NA00b] and in a student project of Stefan Kneubuehl [Kne01].

The paper "Explicit Namespaces" [AN00] explains how forms unify a number of concepts, including namespaces, thereby enabling the definition of abstractions, such as generic wrappers, that are difficult or impossible to define in most programming languages.

JPiccola is an implementation of Piccola3. The PhD thesis of Franz Achermann [Ach02] describes the syntax and semantics of Piccola3 in detail. This language is based on the Piccola calculus, a further refinement of $\pi\mathcal{L}$ and the Form calculus, which provides form introspection, and a number of features such as first-class abstractions as built-in mechanisms.

One of the difficulties in implementing a language like Piccola, is to realize an effective bridge to the host language. This subject is explored in detail in the diploma thesis of Nathanael Schaerli [Sch01, SA01], and a highly effective partial evaluation strategy is developed in the Squeak implementation of SPiccola that drastically reduces the wrapping and unwrapping of host objects.

Achermann proves in his dissertation [Ach02] that this strategy is sound. This strategy is also adopted and refined in the present implementation of JPiccola, and is described in the diploma thesis of Stefan Kneubuehl [Kne03].

Another difficulty when the host language is object-oriented, is that many components cannot be composed without creating a subclass of an existing host-language class. To create a Java GUI, for example, one must subclass a listen class or implement a listener interface. Since Java provides only introspection, and not full reflection, the problem cannot be solved without generating a number of boilerplate Java classes. In his diploma thesis [Sch03], Andreas Schlapbach demonstrates a JPiccola approach in which these boilerplate class can be generated on-the-fly by using a bytecode generation package.

Another open problem is to develop a static type system for Piccola. Lumpe partially explored this topic in his dissertation [Lum99], but only considered typing of $\pi\mathcal{L}$, not Piccola itself. Nierstrasz has proposed an experimental system of "Contractual Types" [Nie03] in the context of a pure form calculus, but has not applied it to Piccola. Stefan Kneubuehl, in his diploma thesis [Kne03], extends contractual types to handle Piccola features not present in the (pure) form calculus, and develops an experimental type inference system for JPiccola.

Cited publications are available from the scg web site:

www.iam.unibe.ch/∼scg/cgi-bin/oobib.cgi?query=piccola.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the following projects:

- "Composing Active Objects" (SNF Project No. 2140610.94, Oct. 1994 - Sept. 1996),
- "Infrastructure For Software Component Frameworks" (SNF Project No. 2000-46947.96, Oct. 1996 - Sept. 1998),
- "A Framework Approach to Composing Heterogeneous Applications", (SNF Project No. 20-53711.98, Oct. 1998 - Sept. 2000),
- "Meta-models and Tools for Evolution Towards Component Systems" (SNF Project No. 20-61655.00, Oct. 2000 - Sept. 2002), and
- "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004).

Many thanks to Gabriela Arévalo, Alexandre Bergel, Markus Gaelli, Tudor Girba, Marc-Philippe Horvath and Laura Ponisio for their suggestions and corrections.

Chapter 6

JPiccola FAQ

Q: Where can I find prelude.picl and all the other standard library scripts?
A: Unpack the JPiccola jar file (jar xvf JPiccola3.6b.jar). prelude.picl is in the lib folder.

Q: How do I run all the standard tests from the console?
A: Run: `(loadCore "test/testAll.picl" root).main()`

Q: How do I print the current stack trace without raising an exception?
A: Run: `Debug.printStackTrace(Debug.getStackTrace(dynamic))`. See Section 3.1.4.

Q: What does it mean when Piccola reports `<No source information available>`?
A: This means that the stack trace is referring to a piece of code that has no source code reference. Mostly, this is code generated by the vm such as Piccola wrappers for Java method calls. Just ignore it.

Q: How do I invoke a Java method that takes an int or a long argument, like `Thread.sleep()`?
A: `Host.class("java.lang.Thread").sleep[(N*1000).asLong()]`

Bibliography

[Ach00] Franz Achermann. Language support for feature mixing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

[Ach02] Franz Achermann. *Forms, Agents and Channels – Defining Composition Abstraction with Style.* PhD thesis, University of Berne, January 2002.

[AKN00] Franz Achermann, Stefan Kneubuehl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[Cri99] Cristina Gheorghiu Cris. Visualisierung von pi-programmen. Informatikprojekt, University of Bern, January 1999.

[Kne01] Stefan Kneubuehl. Implementing coordination styles in piccola. Informatikprojekt, University of Bern, February 2001.

[Kne03] Stefan Kneubuehl. Typeful compositional styles. Diploma thesis, University of Bern, April 2003.

[LAN00] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using metaobjects to model concurrent objects with PICT. In *Proceedings of Languages et Modèles à Objects*, pages 1–12, Leysin, October 1996.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, September 1997.

[Lum99] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition.* Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[NA00a] Oscar Nierstrasz and Franz Achermann. Separation of concerns through unification of concepts. In *ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, 2000.

[NA00b] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, Nov 1-2 2000. IEEE.

[Nie90] Oscar Nierstrasz. A guide to specifying concurrent behaviour with abacus. Object management, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[Nie03] Oscar Nierstrasz. Contractual types. submitted for publication, 2003.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.

[NP90] Oscar Nierstrasz and Michael Papathomas. Viewing objects as patterns of communicating agents. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 38–43, October 1990.

[NSA00] Oscar Nierstrasz, Jean-Guy Schneider, and Franz Achermann. Agents everywhere, all the time. In *ECOOP 2000 Workshop on Component-Oriented Programming*, 2000. Web proceedings available at: http://www.cs.rug.nl/ bosch/WCOP2000/.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing composable software systems – A research agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS '96*, pages 271–282. Chapmann & Hall, 1996.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings Theory and Practice of Parallel Programming (TPPP 94)*, pages 187–215, Sendai, Japan, 1995. Springer LNCS 907.

[SA01] Nathanael Schärli and Franz Achermann. Partial evaluation of inter-language wrappers. In *Workshop on Composition Languages, WCL '01*, September 2001.

[Sch99] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition.* Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[Sch01] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[Sch03] Andreas Schlapbach. Enabling white-box reuse in a pure composition language. Diploma thesis, University of Bern, January 2003.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.