SYLVAIN HALLÉ

EVENT STREAM PROCESSING WITH BEEPBEEP 3

Log crunching and analysis made easy

EVENT STREAM PROCESSING WITH BEEPBEEP 3

Membre de L'ASSOCIATION NATIONALE DES ÉDITEURS DE LIVRES

Presses de l'Université du Québec
Le Delta I, 2875, boulevard Laurier, bureau 450, Québec (Québec) G1V 2M2
Téléphone : 418 657-4399 *Télécopieur :* 418 657-2096
Courriel : puq@puq.ca *Internet :* www.puq.ca

Diffusion / Distribution :

Canada	Prologue inc., 1650, boulevard Lionel-Bertrand, Boisbriand (Québec) J7H 1N7 Tél. : 450 434-0306 / 1 800 363-2864
France et Belgique	Sofédis, 11, rue Soufflot, 75005 Paris, France – Tél. : 01 53 10 25 25 Sodis, 128, avenue du Maréchal de Lattre de Tassigny, 77 403 Lagny, France – Tél. : 01 60 07 82 99
Suisse	Servidis SA, Chemin des Chalets 7, 1279 Chavannes-de-Bogis, Suisse – Tél. : 022 960.95.25

Diffusion / Distribution (ouvrages anglophones) :

Independent Publishers Group, 814 N. Franklin Street, Chicago, IL 60610 – Tel. : (800) 888-4741

The Copyright Act forbids the reproduction of works without the permission of rights holders. Unauthorized photocopying has become widespread, causing a decline in book sales and compromising the production of new works by professionals. The goal of the logo is to alert readers to the threat that massive unauthorized photocopying poses to the future of the written work.

SYLVAIN HALLÉ

EVENT STREAM PROCESSING WITH BEEPBEEP 3

Log crunching and analysis made easy

Presses de l'Université du Québec

Bibliothèque et Archives nationales du Québec and Library and Archives Canada cataloguing in publication

Hallé, Sylvain, author

Event stream processing with BeepBeep 3 : log crunching and analysis made easy / Sylvain Hallé.

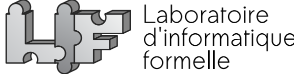
(Mesure et évaluation)

Includes bibliographical references and index.
Issued in print and electronic formats.

ISBN 978-2-7605-5101-5
ISBN 978-2-7605-5102-2 (PDF)

1. Event processing (Computer science). I. Title. II. Series : Collection Mesure et évaluation.

QA76.9.D5H35 2018 004'.36 C2018-943031-1
C2018-943032-X



Graphic Design
Richard Hodgson

Cover Photos
iStock

Legal Deposit : 1st quarter 2019
› Bibliothèque et Archives nationales du Québec
› Bibliothèque et Archives Canada

© 2019 – Presses de l'Université du Québec
Tous droits de reproduction, de traduction et d'adaptation réservés

Printed in Canada
D5101-1 [01]

Contents

Contents iv

Foreword viii

1 Introduction 1

Computations Over Streams . . . 2
Why Use an Event Stream Processing System? . . . 3
What is BeepBeep? . . . 4
Getting Started . . . 7
How to Read This Book . . . 8
Code Examples and Exercises . . . 10
Building BeepBeep . . . 12
Acknowledgements . . . 13

2 Basic Concepts 15

Processors . . . 15
Pulling Events . . . 16
Piping Processors . . . 19
Two Common Mistakes . . . 20
Processors with More than One Input . . . 22
When Types do not Match . . . 25
Pushing Events . . . 27
Pushing on Binary Processors . . . 29

Closing Processor Chains 31
Exercises 33

3 Fundamental Processors and Functions 35
Function Objects 35
Applying a Function on a Stream 37
Function Trees 40
Forking a Stream 43
Cumulating Values 47
Trimming Events 51
Sliding Windows 54
Grouping Processors 58
Decimating Events 62
Filtering Events 64
Slicing a Stream 68
Keeping the Last Event 73
Exercises 77

4 Advanced Features 79
Lists, sets and maps 79
Pumps and Tanks 91
Basic Input/Output 94
Soft vs. Hard Pulling 102
Partial Evaluation 108
The State of a Processor 110
Duplicating Processors 115
Exercises 120

5 The Standard Palettes 123
Tuples 123
Finite-state Machines 130
First-order Logic and Temporal Logic 139
Java Widgets 153
Plots 156
Signal Processing 161
Networking 168
JSON and XML Parsing 176
Exercises 184

6 A Few Use Cases 187

Stock Ticker 187
Medical Records Management 194
Online Auction System 196
Voyager Telemetry 200
Electric Load Monitoring 206
Video Game 212
Exercises 218

7 Extending BeepBeep 219
Creating Custom Functions 219
Create your Own Processor 228

8 Designing a Query Language 245
The Turing tarpit of a Single Language 245
Defining a Grammar 247
Building Objects from the Parsing Tree 250
Simpler Stack Manipulations 256
Building Processor Chains 260
Mixing Types 266

A Drawing Guide 273

B Glossary 279

C Further Reading 303

Index 311

Foreword

This foreword could also be called: *Sylvain, why did you write this book?*

Short answer: because I am *lazy*.

In the research lab where I work at Université du Québec à Chicoutimi (the *Laboratoire d'informatique formelle* or LIF), I have been developing the system described in this book for a couple of years. It began as a small library with just a few hundred lines of code, which was progressively restructured, extended, refactored, split and merged, to become the relatively stable product detailed in the following pages: the **BeepBeep** event stream processing engine.

Over the years, many of my students had their hands on BeepBeep as part of their research projects. Some of them were summer internship undergrads who were asked to develop a specific extension to the existing software. Others were Masters or even PhD students, who provided a deeper contribution to the system, or tested it extensively in R&D projects with industry.

Every time a new colleague would join our team (whether a new student or a new faculty member), the same problem would happen: he or she needed to be taught what BeepBeep was, how it worked, and what it could and couldn't do. Most people in the group had bits and pieces of that information, but for the most part, *I* was the one with an eye on everything. So the task of tutoring newcomers on the nuts and bolts of the system would generally fall on me.

Make no mistake: I love teaching, and I love talking about BeepBeep. However,

I realized over time that this permanent one-on-one coaching could not be sustainable for long. First, there is the time issue: like all good university professors, I tend to put more on my plate than I can actually manage, which leaves little room for regular private lessons. But most importantly, I soon acknowledged that I lacked good *teaching material* about BeepBeep.

Sure, we wrote about half-a-dozen scientific papers about the system in the past four or five years. In the beginning, I assumed I could simply staple these papers together, give them to any BeepBeep neophyte and call it a day. In retrospect, I can see why this doesn't work: a research paper is meant for a technical audience of knowledgeable people, and is very narrow in scope –hardly the layman's gentle introduction to some topic. I came to admit that without a thorough and well-structured tutorial, BeepBeep would still rely on oral tradition in order to be understood by future users (assuming people are still interested in BeepBeep in years to come). People from outside our lab would probably never know it exists, and if they did, would probably never take the time to decrypt the research papers by themselves, let alone make their minds about whether it could be useful to them or not. I had to work on a "BeepBeep book", if I wanted this book to work for me afterwards. Then, I could afford to be lazy.

The rest is simple: I sat down and started typing. This simple process had a positive impact on the system itself: it made me fix some inconsistencies in naming conventions, forced me to standardize and extend the pool of graphical symbols I was already using informally, and overall, added a layer of polish on the library to make it presentable to the outside world. What started as a small documentation file ended up as a complete book, which in its current version contains:

- **135** examples, for a total of 4,500 lines of Java code
- **134** colour illustrations
- **31** exercises across all chapters

Book Toolchain

This book exists in two versions:

- An "e-book" (or PDF) version, published by the Presses de l'Université du Québec (PUQ) and accessible through their web-

site: `https://www.puq.ca`. This book is published under an *open access* policy; it has been given an ISBN and has all the features of a "real" paper book: reviewing, editing, copyright registration. However, it is only accessible electronically –free of charge. **If you want to cite BeepBeep's book in your work, please cite this version.**

- An online, interactive version, accessible on GitBook:
  `https://liflab.gitbook.io`
  This version is viewable in a web browser; contrary to the PUQ book, it *will* be updated in the future to match the evolution of the library. However, I shall stress that the PUQ are not involved in the content of that version.

For those who are curious, both the PDF and the GitBook versions, although they look different, are generated from the same source files, written in the Markdown format. (Maintaining two versions in parallel would be a nightmare.) To this end, I set up a simple toolchain, made of a collection of PHP scripts and Java programs, which can convert the original files into a directory structure suitable for GitBook. What is more, all the source code examples are not hard-coded into the text, but are rather dynamically inserted from references to markers in the actual source code repository –making it much easier to keep the code and the book in sync.

However, if you want to turn that same documentation into a printable version, GitBook it not really appropriate. The file it generates looks like a collection of web pages printed to PDF from a web browser and stacked one after the other. This is acceptable for e-readers or on-screen viewing, but gives a rather substandard look for a printed document. Even bare-bones features, such as page numbers and a table of contents, are absent from the generated document. Besides, apart from a few stylesheet tweaks, you have absolutely no control over the appearance of the resulting PDF. Needless to say, this output cannot be used as the basis for a professional-looking printed book.

This is why I wrote some more scripts to generate, from the same sources, another directory structure where the files are converted to LaTeX, thanks to the Pandoc conversion software. In such a way, a decent stylesheet can be applied to the book, which also benefits from all the usual LaTeX goodies: an index, a table of contents, vector graphics instead of GitBook's bitmaps, spotless typography, and so on. Those interested may have a look at the GitHub repository containing the basic structure of a GitBook/LaTeX hybrid document at: `https://github.com/sylvainhalle/gitbook-latex`.

If you are of the arts-and-crafts type, you can easily print and bind the PDF book by yourself. Its dimensions are exactly that of a US Legal sheet cut in two; simply print the document at two pages per sheet (disable the shrink/fit option to avoid them to be resized) and use your favourite paper cutter! If you print double-sided, however, you must make sure that the correct pages are facing each other (this is trickier than you might think). The source code repository for this book contains a PHP script that produces a correct "2-up" PDF from the original book; it is available at:
`https://github.com/liflab/beepbeep-3-book/blob/master/generate-2up.php`

Long-term preservation of resources

Tim Berners-Lee, inventor of the web, once said: "Cool URIs don't change". Unfortunately, online resources tend to move around, and sometimes vanish.

When I started my undergraduate studies (almost 20 years ago), *SourceForge* was the platform where all cool projects were developed. Today, the site is the shadow of itself, and seems to stay online to host the latest version of legacy software projects. A few years ago, *Google Code* was the new repository to hang around; yet in 2015, Google announced it would close down the platform, leading to the disappearance of thousands of software projects.

BeepBeep is currently hosted on GitHub, a popular and dynamic software repository. But GitHub could have the same fate as the repositories that came and went before it. If you read this book at some time in the future, you may not be able to find the resources at the URLs they are supposed to be.

In such a case, your last resort will probably be to look at the *Software Heritage* platform (`https://softwareheritage.org`). The goal of this UNESCO-backed project is to "collect all publicly available software in source code [and] replicate it massively to ensure its preservation". Among other things, Software Heritage indexes and backs up all well-known code repositories. BeepBeep is there, as is the source code for this book. Just search for "beepbeep" and you shall find multiple repositories and forks of the original piece of software.

Acknowledgements

I end this foreword by acknowledging several people who helped me in the development of BeepBeep in general, and of this book in particular:

- All the students who worked *with* or *on* BeepBeep throughout the years: Asma, Aouatef, Armand, Belkacem, Corentin, Dominic, Eva, Jérôme, Kim, Kun, Luis, Massiva, Mohamed (R and Y), Omar, Paul, Pierre-Louis, Quentin, Rémi, Simon, Sébastien, Stéphanie, Théo and Valentin. They have shown commendable patience for working with an unfinished, unpolished tool, and for being the first beta-testers of just about everything there is inside this system.
- My colleagues at LIF, professors Sébastien Gaboury and Raphaël Khoury, who have been willing to use BeepBeep in some of their own projects and gave it its first big shakedown.
- Ginette Tremblay, for carefully reviewing the manuscript and correcting my English slang.
- The staff at Presses de l'Université du Québec, for being open to this unusual book project and accepting to publish it in their *open access* collection.
- My family and friends, who supported me while I was writing this book –sometimes at the most inopportune moments.

I hope that you will find this book both instructive and enjoyable to read!

Sylvain Hallé
Jonquière, Canada
8/16/2018

1 Introduction

Many computing problems can be viewed as the evaluation of queries over data sources called *streams*. A stream is made of discrete data elements, called *events*; these can be as simple as a single number, or as complex as a special data structure with a large number of fields, a piece of text, or even a picture.

Event streams can be generated from a wide variety of sources. A small temperature sensor that is periodically queried by a system produces a stream of numerical values. A web server log saved to a file contains the latest content of a stream where page requests are recorded. Even your monthly credit card statement is a stream of timestamped payments and expenses. The rate at which such streams produce events can vary widely: your personal credit card stream probably contains no more than a few entries per day; a sensor can emit a temperature reading once per minute, while a very busy web server may log entries thousands of times per second.

In most cases, these streams are not interesting by themselves. Rather, we are more likely to extract various kinds of information from them in order to answer questions about their content. What is the maximum temperature reading over the last day? Have I ever bought something at the grocery store two days in a row? How many times a certain page has been requested this week? Like the streams they refer to, the answer to these questions can be a single number, an interval, a table, a plot, or anything else. Computing the answer to these questions, in a nutshell, is the heart of stream processing.

Stream processing can be found in an extremely wide range of applications, but it is not always named as such. For example, observing the behaviour of a program for testing purposes is often called *runtime verification*, yet the sequence of observations made on the program at various moments in time fits the definition of an event stream very well. Amplifying a feed of raw audio samples can also be seen as a very specific form of event stream processing,

although an audio technician would probably never think about it in this way. Very often, a stream is analyzed and transformed on-the-fly, but this is not even a requirement: hence, reading a pre-recorded sequence of events from some static source also counts as stream processing.

Computations Over Streams

At the onset, event stream processing is like any normal programming activity. Given an input stream, one can write a script or a program in the language of one's choice to perform the desired computation. However, certain hypotheses make event stream processing more complex than simple scripting.

1. As its name implies, the source of an event stream processor is a *stream*. This means that data elements arrive progressively, one event at a time. Accessing these event feeds is often more complex than simply opening a file or connecting to a database.
2. We typically expect an answer to be produced as soon as it can be known; this is called *online* processing. For example, if we want to calculate the average temperature on a window of the past 10 readings, the output value should be computed as soon as those 10 readings have been received. This "streaming" mode of operation is to be contrasted with an *offline* or "batch" mode, where results for all windows of 10 events would be computed and output all at once at the end of the program.
3. A stream unfolds in only one direction: forward. It is generally not possible for a stream processor to rewind an input stream and read previous events a second time. If something must be remembered about the past, it is up to the stream processor to store it somewhere.

Guarantees on the delivery of events in a CEP system can also vary. "At most once" delivery entails that every event may be sent to its intended recipient, but may also be lost. "At least once" delivery ensures reception of the event, but at the potential cost of duplication, which must then be handled by the receiver. In between is perfect event delivery, where reception of each event is guaranteed without duplication. These concepts generally matter only for distributed event processing systems, where communication links between nodes may involve loss and latency.

Some of these hypotheses can sometimes be modified. For example, if one is reading a stream from a pre-recorded file, it is indeed possible to move back-

wards and return to previous events, contrary to what condition #3 stipulates. However, in the general case, processing a sequence of events in a streaming fashion is a little more involved than writing a generic piece of code.

Over the years, various tools, libraries and systems have been developed to help users process event streams. These tools and techniques can roughly be divided into two groups. The first group of software originates from the database community, and includes tools like Cayuga, Borealis, TelegraphCQ, Esper, LINQ, Siddhi, VoltDB and StreamBase SQL. While their input languages vary, most can best be seen as special cases of database query languages, with added support for computation of aggregate functions (average, minimum, etc.) over sliding windows of events (e.g. all events of the last minute). The second group of software, while not labelled specifically as such, comes from the runtime verification community. Indeed, runtime monitors such as Java-MOP, LARVA, LogFire , MarQ, MonPoly, Tracematches , TeSSLA, J-Lo, PQL, PTQL, SpoX and are designed with the purpose of detecting violations of some sequential pattern of events generated by a system in real time.

It was observed in earlier work by the author of this book that these two classes of systems have complementary strengths. The handling of aggregate functions over events provided by CEP tools is notably lacking in virtually all existing runtime monitors. Conversely, monitors generally allow the expression of intricate sequential relationships between events, using finite-state automata or temporal languages, which go far beyond CEP's traditional capabilities.

Why Use an Event Stream Processing System?

An organization may have multiple log repositories at its disposition: execution logs, server logs, and possibly other real-time sources of events. Useful information can be extracted from these logs, which often lies dormant, dispersed across file servers and databases.

A first, natural step to extract and process data consists of writing a bunch of quick crunching scripts in some mainstream programming language. To this end, Python, PHP or Perl can come in handy. However, as time goes by, a tiny script becomes two, which together grow from a few tens of lines to a few hundreds. More often than not, their content is so specific to the current data-crunching task that hardly anything they contain is worth reusing. Since every script is essentially single-use, not much time is spent on testing or

documentation. The end result is a situation similar to Figure 1.1, which shows a proliferation of hack-together, use-once, throw-away scripts.

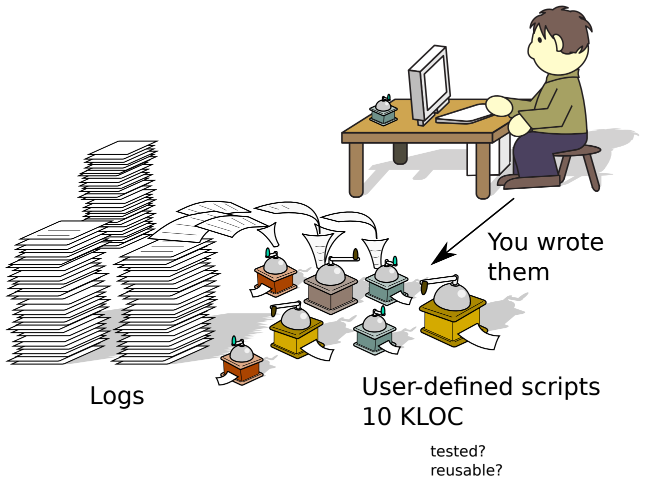


Figure 1.1: Processing logs with user-defined scripts.

In contrast, an event stream processing system (such as BeepBeep) concentrates many recurring log processing tasks in a single location. Users still need to write scripts; however, these scripts can be expressed at a higher level of abstraction, by combining lower-level functions provided by the underlying system. This has for effect of improving their readability, but also of reducing their size. Most importantly, since the functionalities provided by the event stream processing system are intended to be generic and reusable, they are worth spending time to be well documented and tested. As a consequence, the same processing tasks can be accomplished in fewer lines of custom user code. This is what is illustrated in Figure 1.2.

What is BeepBeep?

In this book, you will learn how to use an event stream query engine called BeepBeep to perform various tasks over event streams of different nature. BeepBeep began as an academic research tool developed by the author of this book while he was a PhD student at Université du Québec à Montréal, Canada. Version 1 of the system was developed from 2008 to 2013 and has been the subject of numerous papers and case studies (see the *Further Reading*

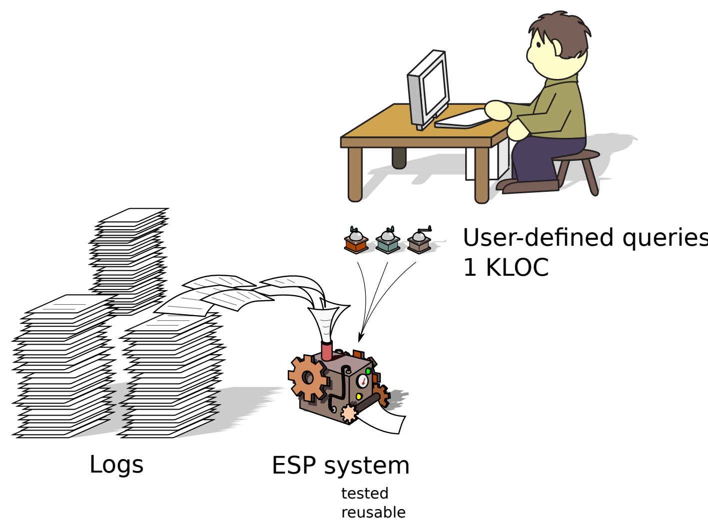


Figure 1.2: Processing logs with an event stream processing system.

section at the end of this book). It was much more limited than the BeepBeep we are talking about in this book, and could only perform a specific kind of stream processing called *runtime verification*. The main distinguishing point of this first version was the handling of complex events with a nested structure (such as XML documents), and an input language that borrowed from a mathematical language called Linear Temporal Logic. BeepBeep 1 is no longer under active development and is considered obsolete for all practical purposes.

In 2013-2014, the version 2 was an attempt at implementing the same concepts as BeepBeep 3. It was cancelled at an early stage of development and was never officially released. One can hence consider BeepBeep 3 as the second "real" incarnation of BeepBeep. It benefits from a complete redesign of the platform, which includes and significantly extends most of the 1.x features.

BeepBeep has a few interesting features distinguishing it from other software systems based on events.

- It is **intuitive**. Virtually every computation in BeepBeep can be expressed in a totally graphical way, using a vast set of pictograms (most of which are detailed in an appendix at the end of this book). Therefore, one does not need to read through Java code to understand a program that uses BeepBeep.

- It is **lightweight**. The core of BeepBeep is a stand-alone Java library that weighs less than 200 kilobytes (yes, that's *kilobytes*). BeepBeep also has low memory requirements; typically, as long as a Java virtual machine is available on a platform, BeepBeep can be made to run on it. It has been used in various environments, ranging from server clusters to smartphones and small devices such as the Raspberry Pi.
- It requires **zero configuration**. To start using BeepBeep, one simply needs to download the library and use the classes it provides in any Java program. Writing a working chain of processors (the basic computing units in BeepBeep) can be done in a few lines of code.
- It **does not force users to use a query language**. Many other event stream processing systems require writing queries in some made-up language vaguely similar to SQL. In contrast, BeepBeep enables users to create, configure and **pipe** processor objects directly. As a result, the computation that is being executed is very close to users' own mental idea of what is happening. (And if users prefer to use a query language, it is possible to create their own; see Chapter 8.)
- It is **modular**. Apart from its small core of basic processors and functions, all other features of BeepBeep are bundled into a large number of optional plug-ins called *palettes*. This is different from many other systems that attempt to provide a huge, monolithic, one-size-fits-all set of functionalities. In BeepBeep, users only use the palettes they need, resulting in a system that carries far less dead code.
- It is **versatile**. There are palettes to read Excel spreadsheets, parse Apache server logs, perform data mining, calculate statistics, analyze network packets, draw plots, and more (see Chapter 6 for some examples). Among the most unusual palettes developed for BeepBeep, one even allows two smartphones to exchange data streams using their onboard camera and QR codes. As long as a problem can be modelled as a form of computation over streams, there is probably a way to do it with BeepBeep.
- It is heavily **customisable**. In case none of the existing palettes meet the users' needs, they can easily create their own processors, functions and events –typically in just a few lines of code (see Chapter 7). These custom-made objects can interact with all the others, meaning you only need to code what is missing, instead of reinventing the wheel.

Although BeepBeep has a host of interesting features, it is not a panacea. There are other things for which it is not as appropriate, or that have been

purposefully excluded from its design:

- It is not a **distributed computing environment**. Although events can easily be passed around across machines using special network palettes, this is a far cry from what elaborate fault-tolerant publish-subscribe dispatching systems can provide.
- It is not a **high-performance computing environment**. Many things can be done reasonably well in BeepBeep, and there are several situations in which it is quite fast. However, if you expect to crunch data at speeds of exabytes per second, chances are BeepBeep will be too slow for your task.

However, if these limitations are not restrictive, BeepBeep can prove an easy and convivial tool to experiment with event stream processing.

Getting Started

BeepBeep is a free and open source software, distributed under the Lesser General Public License (LGPL). Accordingly, its use is free of charge, and the tool may even be included as a library inside commercial software.

In this chapter, you will learn to set up a programming environment using BeepBeep to run the code examples found throughout this book. The set-up instructions use the Eclipse integrated development environment (IDE), but they can easily transfer to other IDEs, or even to a command line-only installation. BeepBeep has very low system requirements, so anything from a Raspberry Pi to a supercomputer should be able to run all the code examples from this book.

The first step is to open an Eclipse workspace, and to create a new empty Java project. BeepBeep must then be downloaded and included into the project. Pre-compiled releases of BeepBeep can be downloaded directly from BeepBeep's GitHub repository (https://github.com/liflab/beepbeep-3), under the *Releases* page. Official releases are stable and well-tested, although the API between releases (especially the old ones) can change slightly. As a rule, there is no good reason not to use the latest release when starting a project.

BeepBeep is made of a single Java archive (JAR) file, called `beepbeep-3.jar`. This file is runnable and stand-alone, or can be used as a library, so it can be moved around to the location of your choice. If you want to create a Java

project that uses BeepBeep, simply include `beepbeep-3.jar` in your CLASSPATH and you are ready to begin. In Eclipse, this means opening the *Build Path* dialog, selecting *Add external JARs*, and pointing to the location of `beepbeep-3.jar` on your machine.

To make sure that everything works, create a new Java class with a `main()` method, and type the following:

```
import ca.uqac.lif.cep.tmf.*;
public class HelloWorld {
  public static void main(String[] args) {
    QueueSource q = new QueueSource();
    q.setEvents("foo");
    System.out.println(q.getPullableOutput().pull());
  }
}
```

✎

This program creates a new instance of a `QueueSource` object, and pulls one event from its output. If everything compiles, and running the program prints a single line with the text `foo`, then the environment is correctly setup to use BeepBeep.

Palettes are additional JAR files that provide complementary functionalities to BeepBeep. Most of the palettes that will be used in this book can be downloaded from a sibling palette repository, located at https://github.com/liflab/beepbeep-3-palettes. The *Releases* page of this repository offers a large zip file, inside which each individual palette is a single JAR file. Palettes can be loaded into a project in the same way as BeepBeep's main JAR file. Note that palettes are not stand-alone: your project still requires `beepbeep-3.jar` even if palettes are included into it. For this reason, palettes are also sensitive to the version of the main JAR that you are using; attempting to load a palette compiled for an older version of BeepBeep may create errors, and vice versa. No problems should occur if the latest versions are used.

How to Read This Book

The first part of this book (chapters 2 to 5) is organized in a roughly linear fashion: each chapter builds on notions that have been covered in the previous one.

- Chapter 2 describes the very basic concepts of BeepBeep's operation: streams, pipes, processors, pushing and pulling events, and composition. You will not understand anything of the rest of this book before first going through this chapter!
- Chapter 3 describes the general-purpose `Function` and `Processor` objects that are provided in the system's core. You will learn how to trim, filter and slice event streams, apply functions and sliding windows to events, and so on. Virtually any BeepBeep program involves one of the objects described in this chapter.
- Chapter 4 describes some more functions and processors specific to particular use cases, such as processing character strings or manipulating collections of objects. It also gives more details about more technical features of `Processor` objects, such as how to copy them, or call their functions across multiple threads.
- Chapter 5 leaves BeepBeep's core, and describes the functionalities provided by a standard set of palettes that have been developed alongside the main software. Not all palettes may be interesting to you, so each section of this chapter is written so as to be relatively independent of the others.

The second part of the book (chapters 6 to 8) is made of independent chapters covering other aspects of BeepBeep.

- Chapter 6 mixes all the content of the previous chapters together, and shows a number of more complex use cases that illustrate the capabilities of BeepBeep and its standard palettes. You will learn how BeepBeep can be used to perform runtime monitoring in a video game, process telemetry from a space probe, or analyze the power consumption of home appliances, among other things.
- Chapter 7 is intended for BeepBeep developers. It shows how Java programmers can easily create their own `Processor` and `Function` objects, package them into their own palette, and make them interact with other BeepBeep objects.
- Chapter 8 concentrates on one particular BeepBeep palette, called *DSL*. Rather than piping processors directly using Java, this palette makes it possible for end-users to define the syntax of a custom language, and to write an *interpreter* that builds processor chains automatically from expressions of that language.

Finally, the book ends with a few appendices that are meant as a reference.

- Appendix A defines the broad guidelines for drawing processor chains similar to the illustrations shown throughout this book.
- Appendix B is an illustrated glossary listing all the `Processor` and `Function` objects provided by BeepBeep and its palettes, and which are mentioned somewhere in the book. For each of them, it shows the standard picture used to represent them and provides a short definition.
- Appendix C is a list of references to books and scientific papers providing more details about some of the topics discussed in this book.

Code Examples and Exercises

Most of the code examples in this book are also available online in a single big project. This project can be downloaded from GitHub at https://github.com/liflab/beepbeep-3-examples. It contains an extensive Javadoc documentation of every file, which can be explored online at https://liflab.github.io/beepbeep-3-examples.

When a code snippet is followed by the ✎ symbol, this indicates that this piece of code is also available online in the code example repository. When viewing an electronic version of this book (such as an online website or a PDF), the ✎ symbol is actually a hyperlink leading directly to the first line of that snippet in the GitHub repository. As an example, try clicking on the link corresponding to the following code block:

```
QueueSource source = new QueueSource();
source.setEvents(1, 2, 4, 8, 16, 32);
Pullable p = source.getPullableOutput();
for (int i = 0; i < 8; i++)
{
    int x = (Integer) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

We can also notice that the online version of the code is sometimes interspersed with comment lines that are absent from the book examples. This is done to improve the legibility of the examples, given that they are already discussed at length in the text itself.

At the end of each main section, a few coding exercises are also suggested.

These exercises require the creation of chains of processors performing specific tasks. Writing an exercise all by yourself, and moving on to the next one, would be a bit pointless. It is possible to determine whether exercises have been done correctly by testing them into a self-grading program called the **tutor**.

The program can be downloaded from https://github.com/liflab/beepbeep-3-tutor. It comes in the form of a single file, called `beepbeep-3-tutor.jar`, which can be integrated in a project like all the other JARs mentioned earlier. This library exposes an object called `Tutor`. All exercises in the book have a unique name; for example, exercise number 2 of Chapter 2 is called `C2E2`. There exist tutor instances for many exercises; you can get the instance of your choice through `Tutor`'s static method `get()`. If a tutor does not exist for an exercise, an exception will be thrown. (At the time of this writing, the tutor is still a work in progress.)

In order to check the tutor setup, it is possible to request a dummy `Tutor` object for an exercise named `TEST`:

```
Tutor tutor = Tutor.get("TEST");
```

The correct answer to this exercise is a single `Processor` object that lets all events pass through; this is done by the aptly named `Passthrough` processor. To let the tutor check the answer, it has to be told what are the inputs and the outputs of this processor chain:

```
Passthrough pt = new Passthrough();
tutor.setInput(pt).setOutput(pt);
```

The tutor feeds events through the input of the chain of processors, and observes what comes out of the output. The tutor can then be asked to check the solution through method `check`:

```
tutor.check();
```

By running the program, after some time, the tutor should print at the terminal:

```
Looks like everything is OK!
```

To show what happens when a solution is incorrect, we shall now give the tutor a modified chain of processors, which discards the first input event. This is done by the `Trim` processor.

```
Trim tr = new Trim(1);
tutor.setInput(tr).setOutput(tr);
tutor.check();
```

Running this program will produce an output as the following:

```
I found an error in your solution.
* With the input trace "A", "B", ...
  I got the output "B" at position 0
  I expected "A".
```

This indicates that the tutor found an input stream for which the output does not match what is expected of the correct solution. Here, since the `Trim` processor discards the first event it receives, the first event to be output is the letter “B” instead of the expected “A”.

Building BeepBeep

Instead of using a pre-compiled release, users may want to build BeepBeep directly from the sources, thus giving access to the very latest features. First make sure the following has been installed:

- The Java Development Kit (JDK) to compile. BeepBeep is developed to comply with Java version 6; it is probably safe to use any later version.
- Ant to automate the compilation and build process

Download the sources for BeepBeep from GitHub or clone the repository using Git:

```
git@github.com:liflab/beepbeep-3.git
```

The project has a few dependencies; any libraries missing from the system can be automatically downloaded by typing:

```
ant download-deps
```

This will put the missing JAR files in the `deps` folder in the project’s root. The sources can be compiled by simply typing:

```
ant
```

This will produce a file called `beepbeep-3.jar` in the folder. In addition, the script generates in the `doc` folder the Javadoc documentation for using BeepBeep.

BeepBeep can also test itself by running:

```
ant test
```

Unit tests are run with jUnit; a detailed report of these tests in HTML format is available in the folder `tests/junit`, which is automatically created. Code coverage is also computed with JaCoCo; a detailed report is available in the folder `tests/coverage`.

For the sake of clarity, we give below the hashes of the latest commits on the various GitHub repositories containing BeepBeep code and examples. All the examples in this book are based on the software in the state it was when these commits were pushed:

- BeepBeep core: `14bb8359fabd507642235e81e88e546e0669f7d5`
- BeepBeep palettes: `b99ca2f8c6b897821a1e54b73b5bae0fa2dc3214`
- Code examples: `27043f6175b54b06bf6d17064bcabecc22c9e9ec`

Acknowledgements

This work was done thanks to the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canada Research Chair on Software Specification, Testing and Validation.

2 Basic Concepts

This chapter is about the fundamental principles for using BeepBeep through simple examples. More specifically, instructions will be provided for the basic usage of processors and functions, two of the most important objects the system provides.

Processors

The first fundamental building block of BeepBeep is an object called a **processor**. This object takes one or more *event streams* as its input, and returns one or more *event streams* as its output. A processor is a stateful device: for a given input, its output may depend on events received in the past. Virtually all the processing of event traces is done through the action of a processor, or a combination of multiple processors chained together to achieve the desired functionality. In terms of Java, all processors are descendants of the generic `Processor` class.

An easy way to understand processors is to think of them as "boxes" having one or more "pipes". Some of these pipes are used to feed events to the processor (input pipes), while others are used to collect events produced by the processor (output pipes). Throughout this book, processors will often be represented graphically exactly in this way, as in the upcoming diagram. A processor object is represented by a square box, with a pictogram giving an idea of the type of computation it executes on events. On the sides of this box are one or more "pipes" representing its inputs and outputs. Input pipes are indicated with a red, inward-pointing triangle, while output pipes are represented by a green, outward-pointing triangle.

The colour of the pipes themselves will be used to denote the type of events

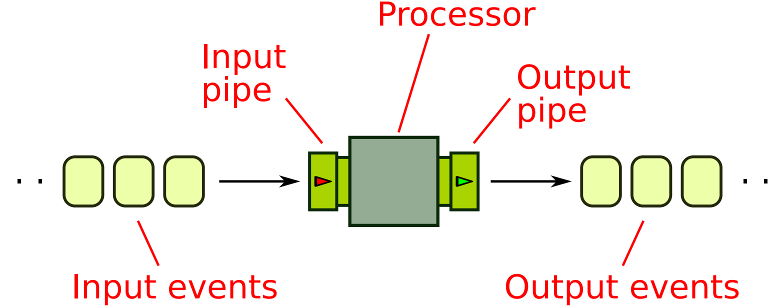


Figure 2.1: A graphical representation of a generic processor taking one input stream, and producing one output stream.

passing through them. According to the convention in this book, a blue-green pipe represents a stream of numbers; a grey pipe contains a stream of Boolean values, etc.

The number of input and output pipes is called the (input and output) **arity** of a processor; these two numbers vary depending on the actual type of processor we are talking about. For example, the previous diagram represents a processor with an input arity of 1, and an output arity of 1. Events come in by one end, while other events (maybe of a different kind) come out by the other end.

A processor produces its output in a *streaming* fashion: this means that output events are made available progressively while the input events are consumed. In other words, a processor does not wait to read its entire input trace before starting to produce output events. However, a processor can require more than one input event to create an output event, and hence may not always output something right away.

Pulling Events

There are two ways to interact with a processor. The first is by getting a hold of the processor's output pipe, and by repeatedly asking for new events. The action of requesting a new output event is called **pulling**, and this mode of operation is called *pull mode*.

Let us instantiate a simple processor and pull events from it. The following

code snippet shows such a thing, using a processor called `QueueSource`.

```
QueueSource source = new QueueSource();
source.setEvents(1, 2, 4, 8, 16, 32);
Pullable p = source.getPullableOutput();
for (int i = 0; i < 8; i++)
{
    int x = (Integer) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

The `QueueSource` object is a simple processor that does only one thing. When it is created, it is given a list of events; from that point on, it will endlessly output these events, one by one, looping back at the beginning of the list when it reaches the end. The first two lines of the previous snippet create a new instance of `QueueSource`, and then give the list of events it is instructed to repeat (in this case, the events are integers). Graphically, this can be represented as follows:

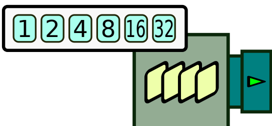


Figure 2.2: A first example.

As one can see, the `QueueSource` object is a special type of processor that has an output pipe, but no input pipe (that is, its input arity is zero). This means that it does not produce events based on the output produced by other processors; in other words, it is impossible to connect another processor into a `QueueSource` (or into any other processor of input arity zero, for that matter). Rather, output events are produced "out of thin air" –or more accurately, from a list of values that is given to the source when it instantiated. In the diagram, this list is shown in the white rectangle overlapping the source's box.

To collect events from a processor's output, one uses a special object called a `Pullable`. The third instruction takes care of obtaining an instance of `Pullable` corresponding to `QueueSource`'s output, using a method called `getPullableOutput()`.

A `Pullable` can be seen as a form of iterator over an output trace. It provides a

method, called `pull()`; each call to `pull()` asks the corresponding processor to produce one more output event. The loop in the previous code snippet amounts to calling `pull()` eight times. Since events handled by processors can be anything (Booleans, numbers, strings, sets, etc.), the method returns an object of the most generic type, i.e. `Object`. It is up to the user of a processor to know what precise type of event this return value can be cast into. In our case, we know that the `QueueSource` we created returns integers, and so we manually cast the output of `pull()` into objects of this type.

Since the queue source loops through its array of events, after reaching the last (32), it will restart from the beginning of its list. The expected output of this program is:

```
The event is: 1
The event is: 2
The event is: 4
The event is: 8
The event is: 16
The event is: 32
The event is: 1
The event is: 2
```

Note that `source` springs into action only upon a call to `pull()` on its `Pullable` object. That is, it computes and returns a new output event only upon request. In other words, we can see it as some kind of gearbox that does something only when we turn the crank: each turn of "crank" triggers the production of a new output event.

As a final note, the Pullable interface extends the Java `Iterator` and `Iterable` interfaces. This means that an instance of Pullable can also be iterated over like this:

```
Pullable p = ...;
for (Object o : p)
{
  // Do something
}
```

This simple example shows the basic concepts around the use of a processor:

- An instance of a processor is first created
- To read events from its output, we must obtain an instance of a `Pullable` object from this processor

- Events can be queried by calling `pull()` on this `Pullable` object

Piping Processors

BeepBeep provides dozens of processors, but each of them in isolation performs a simple operation. To perform more complex computations, processors can be composed (or "piped") together, by letting the output of one processor be the input of another. This piping is possible as long as the type of the first processor's output matches the type expected by the second processor's input.

Let us create a simple example of piping by building upon the previous example, as follows:

```
QueueSource source = new QueueSource();
source.setEvents(1, 2, 3, 4, 5, 6);
Doubler doubler = new Doubler();
Connector.connect(source, doubler);
Pullable p = doubler.getPullableOutput();
for (int i = 0; i < 8; i++)
{
    int x = (Integer) p.pull();
    System.out.println("The event is: " + x);
    UtilityMethods.pause(1000);
}
```

✎

First, a `QueueSource` is created as before; then, an instance of another processor called `Doubler` is also created. For the sake of the example, let us simply assume that `Doubler` takes arbitrary integers as its input, multiples them by two, and returns the result as its output.

The next instruction uses the `Connector` object to pipe the two processors together. The call to method `connect()` sets up the processors so that the output of `source` is sent directly to the input of `doubler`. Graphically, this can be represented as follows:

Notice how the diagram now contains two boxes: one for the source, and one for the doubler. The call to `Connector`'s `connect` is represented by the "pipe" that links the output of the source to the input of the doubler.

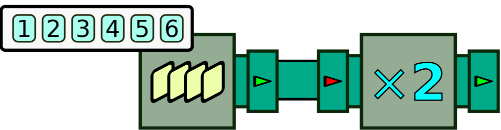


Figure 2.3: Piping the output of a `QueueSource` into a `Doubler` processor.

We can then obtain `doubler`'s `Pullable` object, and fetch its output events like before. The output of this program will be:

```
The event is: 2
The event is: 4
The event is: 6
The event is: 8
...
```

As expected, each event of the output stream is the double of the one at matching position in the source's input stream.

Notice how we obtained a hold of `doubler`'s output Pullable, and made our `pull` calls on *that* object –not on `source`'s. It is up to the downstream processor to call `pull` on any upstream processors it is connected to, if needed. Concretely, this is what happens:

1. A call to `pull` is made on `doubler`'s `Pullable` object
2. In order to produce an output event, `doubler` needs a new input event. It calls `pull` on `source`'s `Pullable` object
3. Processor `source` produces a new event, and emits it as the return value to its call on `pull`
4. Processor `doubler` now has a new input event; it multiplies it by two, and emits it as the return value to its own call on `pull`

Two Common Mistakes

This simple example of processor piping brings us to talk about two common mistakes one can make when creating processors and connecting them.

The first mistake is to **forget to connect two processors**. Suppose that in the original `Doubler` example, we omit the call to `connect`, resulting in a diagram that looks like the following:

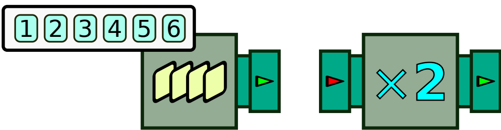


Figure 2.4: Forgetting to pipe two processors.

Notice how the pipe between the source and the Doubler processor is missing. Attempting to call `pull` on `doubler` will throw an exception. The expected output of the program should look like this:

```
Exception in thread "main" ca.uqac.lif.cep.Pullable$PullableException:
Input 0 of this processor is connected to nothing
  at ca.uqac.lif.cep.SynchronousProcessor$OutputPullable.hasNext...
```

What happens concretely is that, when a call to `doubler`'s `Pullable` object is made, it turns around to ask for an input event from upstream, and realizes that it has never been told whom to ask (this is what the call to `connect` does). Consequently, it throws a `PullableException` alerting the user of that issue. Notice that this is a *runtime* error; the program still compiles perfectly.

The second mistake is to **call `pull` on an intermediate processor**. In our original example, it would be an error for the user to perform their own pulls on `source`, instead of or in addition to the pulls on `doubler`. Consider the same chain of processors as above, but with the loop replaced by the following instructions:

```
Pullable p = doubler.getPullableOutput();
System.out.println("The event is: " + p.pull());
System.out.println("The event is: " + p.pull());
Pullable p2 = source.getPullableOutput();
System.out.println("The event is: " + p2.pull());
System.out.println("The event is: " + p.pull());
```

✎

Notice how, after performing two pulls on `doubler`'s `Pullable`, we perform one call to `source`'s `Pullable`, and then resume pulling on `doubler`. The output of this program is this:

```
The event is: 2
The event is: 4
The event is: 3
```

```
The event is: 8
```

The first two outputs are identical to our original program. As was just explained, the first two calls to `pull` on `doubler` resulted in the background in two other calls to `pull` on `source`. The third line corresponds to the pull on `source` directly; it outputs the third event of its list, which is 3. However, since we pulled on `source` directly, that event never reaches the input of `doubler`. As far as `source` is concerned, its third event has duly been sent, and it moves on to the next. Therefore, when calling `pull` again on `doubler`, `source` sends it its *fourth* event (the integer 4), and hence the next line of the output is 8.

As one can see, it generally does not make much sense to pull on processors that are not at the very end of the chain. To prevent the possibility of mistakes, it is possible to encapsulate a group of processors into a "box" that only gives access to the very last `Pullables` of a chain –on which we will elaborate later.

Processors with More than One Input

We mentioned earlier that processors can have more than one input "pipe", or one or more output "pipe". The following example shows it:

```
QueueSource source1 = new QueueSource();
source1.setEvents(2, 7, 1, 8, 3);
QueueSource source2 = new QueueSource();
source2.setEvents(3, 1, 4, 1, 6);
Adder add = new Adder();
Connector.connect(source1, 0, add, 0);
Connector.connect(source2, 0, add, 1);
Pullable p = add.getPullableOutput();
for (int i = 0; i < 5; i++)
{
    float x = (Float) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

This time, we create *two* sources of numbers. We intend to connect these two sources of numbers to a processor called `add`, which, incidentally, has two input pipes. The interesting bit comes in the calls to `connect()`, which now include a few more arguments. The first call connects the output of `source1`

to the *first* input of a processor called `add`. The second call connects the output of `source2` to the *second* input of `add`. Graphically, this is represented as follows:

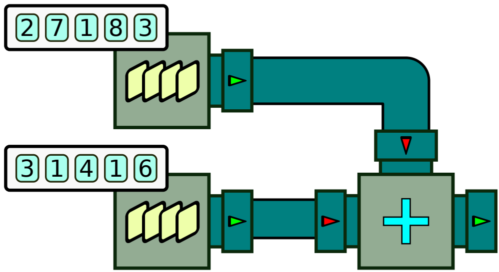


Figure 2.5: A processor with an input arity of two.

In a program, the two input pipes of `add` can be easily accessed through their number (0 or 1). However, in a drawing, it may be difficult to decide which is which. In this book, we will follow the convention that the topmost input pipe is that with the lowest number. Hence, in the previous diagram, input pipe 0 is the one connected to the queue 2-7-1-8-3. In some diagrams, this may still not be clear enough; in such cases, we will explicitly write numbers next to the pipes to tell them apart.

The rest of our program is done as usual: a `Pullable` is obtained from `add`, and its first few output events are printed:

```
The event is: 5.0
The event is: 8.0
The event is: 5.0
The event is: 9.0
...
```

The previous example shows that the output of `add` seems to be the pairwise sum of events from `source1` and `source2`. This is, in fact, exactly the case: 2+3=5, 7+1=8, 1+4=5, and so on. When a processor has an input arity of 2 or more, it processes its inputs in batches called **fronts**. A *front* is a set of events in identical positions in each input trace. Hence, the pair of events 2 and 3 corresponds to the front at position 0; the pair 7 and 1 corresponds to the front at position 1, and so on.

When a processor has an arity of 2 or more, the processing of its input is generally done *synchronously*. This means that a computation step will be

performed if and only if a new event can be consumed from each input stream. It this is not the case, the processor **waits** (and the call to `pull` blocks) until a complete front is ready to be processed. This can be exemplified in the following code example:

```
SlowQueueSource source1 = new SlowQueueSource();
source1.setEvents(2, 7, 1, 8, 3);
QueueSource source2 = new QueueSource();
source2.setEvents(3, 1, 4, 1, 6);
Adder add = new Adder();
Connector.connect(source1, 0, add, 0);
Connector.connect(source2, 0, add, 1);
Pullable p = add.getPullableOutput();
for (int i = 0; i < 5; i++)
{
    float x = (Float) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

The chain of processors in this example is almost identical to the previous example, and can be represented graphically as:

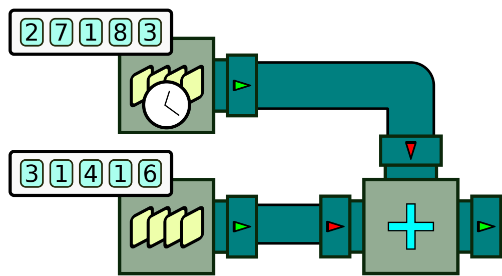


Figure 2.6: A fast source and a slow source.

The difference is that the first queue source has been replaced by a “slow” queue source, which waits 5 seconds before outputting each event. This is represented by the little “clock” in the topmost source box. The output of this program is identical to the previous one:

```
The event is: 5.0
The event is: 8.0
...
```

However, a new line is only printed every five seconds. This can be explained as follows: when a call to `pull` is made on `add`'s `Pullable` object, the processor checks whether a complete front can be consumed. It asks both `source1` and `source2` for a new event; `source2` responds immediately, but `source1` takes five seconds before producing an event. In the meantime, `add` can do nothing but wait. The whole process repeats itself upon every subsequent call to `pull`. Note that `add` only asks for *one* new event at a time from each source; that is, it does not keep pulling on `source2` while it waits for an answer from `source1`.

Synchronous processing is a strong assumption; many other stream processing engines allow events to be processed asynchronously, meaning that the output of a query may depend on what input stream produced an event first. One can easily imagine situations where synchronous processing is not appropriate. However, in use cases where it is suitable, assuming synchronous processing greatly simplifies the definition and implementation of processors. The output result is no longer sensitive to the order in which events arrive at each input, or to the time it takes for an upstream processor to compute an output (the order of arrival of events from the same input trace, obviously, is preserved). Since the timing of arrival of events is irrelevant to the result of a computation, this means that one can perform a "pen and paper" calculation of a chain of processors, and arrive at the same output as the real one, given knowledge of the contents of each input stream.

When Types do not Match

We mentioned earlier that any processor can be piped to any other, *provided that they have matching types*. The following code example shows what happens when types do not match:

```
QueueSource source = new QueueSource();
source.setEvents(3);
Processor av = new ApplyFunction(Numbers.absoluteValue);
Connector.connect(source, av);
Processor neg = new ApplyFunction(Booleans.not);
Connector.connect(av, neg);
System.out.println("This line will not be reached");
```

✎

The culprit lies in the next-to-last line of the program; this is due to the fact that

processor av sends out events of type Number as its output, while processor neg expects events of type Boolean as its input. This can be illustrated as follows:

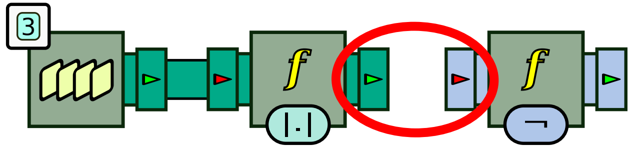


Figure 2.7: Piping processors whose input/output types do not match.

This is the first of our examples using colour coding to represent the type of each stream. Note how number streams and pipes are shown in turquoise, while Booleans are represented using a greyish shade of blue. Using such a graphical representation, the problem can easily be detected: the call to connect attempts to link a turquoise output pipe to a grey-blue input pipe.

Since numbers cannot be converted into Booleans, the call to connect() will throw an exception similar to this one:

```
Exception in thread "main"
ca.uqac.lif.cep.Connector$IncompatibleTypesException:
Cannot connect output 0 of ABS to input 0 of !: incompatible types
    at ca.uqac.lif.cep.Connector.checkForException(Connector.java:268)
    ...
```

Here “ABS” and “!” are the symbols defined for av and neg, respectively. As with the PullableException discussed earlier, this is a *runtime* error. Processor inputs and outputs are not statically typed, so the above program compiles without problem. The error is only detected when the program is being executed, and the Connector object realizes that it is being asked to link processors of incompatible types.

A processor can be queried for the types it accepts for input number n by using the method getInputType(); likewise for the type produced at output number n with getOutputType().

Pushing Events

Earlier we mentioned there were two ways to interact with a processor. The first, which we have used so far, is called *pulling*. The second, as you may guess, is called **pushing**, and works more or less in reverse. In so-called *push mode*, rather than querying events form a processor's output, we give events to a processor's input. This has for effect of triggering the processor's computation and "pushing" events (if any) to the processor's output.

Let us instantiate a simple processor and push events to it. The following code snippet shows such a thing, using a processor called `QueueSink`.

```
QueueSink sink = new QueueSink();
Pushable p = sink.getPushableInput();
p.push("foo");
p.push("bar");
Queue<Object> queue = sink.getQueue();
System.out.println("Events in the sink: " + queue);
queue.remove();
p.push("baz");
System.out.println("Events in the sink: " + queue);
```

✎

The `QueueSink` object is a simple processor that merely accumulates into a queue all the events we push to it. The first line of the previous snippet creates a new instance of `QueueSink`. Graphically, this can be represented as follows:

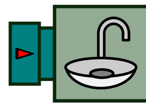


Figure 2.8: Pushing events to a `QueueSink`.

In order to push events to this processor, we need to get a reference to its input pipe; this is done with method `getPushableInput()`, which gives us an instance of a `Pushable` object. A `Pushable` defines one important method, called `push()`, allowing us to give events to its associated processor. In the previous code snippet, we see two calls to method `push`, sending the strings "foo" and "bar".

As we said, `QueueSink` simply accumulates the pushed events into a queue. It

is possible to access that queue by calling a method called `getQueue()` on the processor, as is done on line 5. The contents of that queue are then printed; at this point in the program, the queue contains the strings "foo" and "bar", resulting in the first line printed at the console:

```
Events in the sink: [foo, bar]
```

We then pop the first event of that queue, and then push a new string ("baz") to the sink. The second line the program prints shows the content of the sink at that moment, namely:

```
Events in the sink: [bar, baz]
```

We have seen that calls to `pull` on a processor may result in calls to `pull` on upstream processors. In the same way, in push mode, calls to `push` may result in calls to `push` on downstream processors. In the following code snippet, we connect a `Doubler` processor to a special type of sink, called `Print`, which simply prints to the console every event it receives.

```
Doubler doubler = new Doubler();
Print print = new Print();
Connector.connect(doubler, print);
Pushable p = doubler.getPushableInput();
for (int i = 0; i < 8; i++)
{
    p.push(i);
    UtilityMethods.pause(1000);
}
```

✎

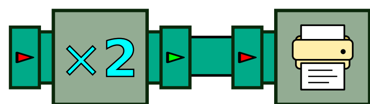


Figure 2.9: Pushing events to a `Print` processor.

The `for` loop pushes the integers 0 to 7 into the input pipe of `doubler`; the `pause` method causes the loop to wait one second (1,000 milliseconds) between each call to `push`. The output of this program is, unsurprisingly, the following:

```
0,2,4,6,8,10,12,14,
```

Notice the one-second interval between each number. This shows that, in push mode, nothing happens until an upstream call to `push` triggers the chain of computation.

Pushing on Binary Processors

The push mode exhibits a special behaviour in the case where a processor has an input arity of 2 or more. Consider the following piece of code:

```
Adder add = new Adder();
Print print = new Print().setSeparator("\n");
Connector.connect(add, print);
Pushable p0 = add.getPushableInput(0);
Pushable p1 = add.getPushableInput(1);
```

✎

This sets up an `Adder` processor, whose output is connected to a `Print` processor, as illustrated below:

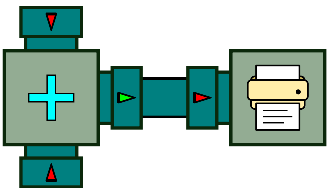


Figure 2.10: Pushing events into a processor of input arity 2.

Since `add` is of input arity 2, it has two `Pushable` objects, numbered 0 and 1. We use a different version of method `getPushableInput()`, which takes an integer as an argument. By convention in our diagrams, the first Pullable object is generally located at the top (or the left) of the box, and the second at the bottom (or the right). Small numbers are located next to the pipes when the context is not clear.

Let us see what happens when pushing numbers into `p0` and `p1`.

```
p0.push(3);
System.out.println("This is the first printed line");
p1.push(1);
p1.push(4);
```

```
p1.push(1);
System.out.println("This is the third printed line");
p0.push(5);
p0.push(9);
```

✎

The first line pushes the number 3 into `p0`. However, since nothing has yet been pushed into `p1`, the first front of events is not complete; `add` is not ready to compute an addition, and nothing is pushed to the printer. This means that the next statement, which prints "This is the first printed line", is indeed the first line to be printed at the console. We then push the number 1 into `p1`; now a complete front is ready to be processed, and `add` pushes the number 4 (3+1) to its output pipe. This event is received by the printer, which prints it at the console.

The next two instructions push two numbers into `p1`. Again, nothing is printed, since no new events have been received from `p0`. Hence the next instruction, "This is the third printed line", does produce the third line of the output. We then push the number 5 into `p0`. An event was already pushed into `p1`, so `add` is ready to compute a new addition, and outputs 9 (4+5).

Here, an important observation must be made. The `add` processor computed the addition of number 5 pushed into `p0` with the number **4** that was pushed earlier into `p1`. That is, a processor consumes events from the same pullable **in the order they arrive**. Number 4 is the second event pushed into `p1`; hence it is matched with the second event pushed into `p0`.

The last instruction pushes the number 9 into `p0`. The third event pushed into `p0` is matched with the third event pushed into `p1`, and `add` pushes 10 (1+9) to its output pipe. The end result of this program should be this:

```
This is the first printed line
4
This is the third printed line
9
10
```

The observation made on the order of arrival is important. It means that each processor has **input queues** to buffer events pushed from upstream until they can be consumed. Here, the numbers 4 and 5 were put into an event queue associated to `p1`, until events were pushed into `p0` and made a computation possible. The nice thing about BeepBeep is that you don't have to worry

about these buffers: the system takes care of them on its own in a completely transparent manner.

Note that the order in which events at the same position in two different streams are pushed still does not matter, though. That is, assuming that we are at the start of the program, writing this:

```
p0.push(3);
p1.push(2);
```

will produce the same output as writing this:

```
p1.push(2);
p0.push(3);
```

The next question that generally comes to one's mind is: what happens if we keep pushing events on only one of the pushables, and nothing on the other? Since no computation can be made, won't this fill the first Pushable's queue endlessly? The answer to this question is simple: yes. If we keep pushing events on only one pushable (or more likely, if one of the upstream sources pushes events much faster than the other), we may end up filling one of the event queues and run out of memory.

Fortunately, there are many use cases (especially realistic ones) where such a catastrophic scenario never occurs. Notice also that this is not a limitation on BeepBeep's side: if the goal is to add numbers from two input streams, and the first generates them at twice the speed of the second, those excess numbers *must* be stored somewhere, and that storage *must* increase linearly with time. There is no escaping it, whether using BeepBeep or not!

Closing Processor Chains

We mentioned earlier that a common mistake is to forget to connect two processors. A variant of this mistake is to forget to attach sources or sinks to the endpoints of a processor chain. Take the very simple example of the `Passthrough` processor, which simply takes input events and returns them as is to its output pipe. It can be depicted as in Figure 2.11.

Let us create a `Passthrough`, and call `pull` on it, as in the following code example.

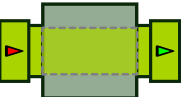


Figure 2.11: A passthrough without a source or a sink.

```
Passthrough passthrough = new Passthrough();
Pullable p = passthrough.getPullableOutput();
p.pull();
```

✎

This program will throw a `ConnectorException` for the same reasons as before: `passthrough` is asked for a new output event, but it is not connected to anything upstream. That makes sense. What is more surprising is that the reverse mistake also exists in push mode. Consider the following example:

```
Passthrough passthrough = new Passthrough();
Pushable p = passthrough.getPushableInput();
p.push("foo");
```

✎

This time, we attempt to push a string ("foo") into `passthrough` –but again, this will throw a `ConnectorException`. Indeed: passthrough is requested to relay an event downstream, but nothing is connected to its output pipe. In the same way events cannot be created out of thin air (in pull mode), they cannot vanish into thin air either (in push mode). In other words, a chain of processors must always be **closed**:

- In pull mode, all upstream endpoints must be connected to a source
- In push mode, all downstream endpoints must be connected to a sink

If, for whatever reason, you want to discard events from a downstream processor, you must still connect it to a sink. However, there is a special sink, called `BlackHole`, which does exactly that.

With these code examples, you know almost everything there is to know about processors in BeepBeep. We have seen how a few simple processor objects can be instantiated and piped together by means of the `Connector` object. We have also explored the two modes by which events can be passed around: *pull* mode where output events are queried by the user, and *push* mode where

input events are produced by the user. We also studied the principles of synchronous processing, and the fact that processors manage internal queues to make sure they always process events at matching positions in their input streams.

Exercises

1. Using the `QueueSource` and `Doubler` processors shown in this chapter, create a chain of processors that outputs the quadruple of the first five odd numbers. As a result, the first five output events should be 4, 12, 20, 28, 36.

2. Using the `QueueSource` and `Add` processors shown in this chapter, create a chain of processors that outputs the sum of each two consecutive prime numbers. The first six prime numbers are 2, 3, 5, 7, 11 and 13. As a result, the first five output events should be 5, 8, 12, 18, 24. (Hint: you will need two `QueueSource`s.)

3 Fundamental Processors and Functions

BeepBeep is organized along a modular architecture. The main part of BeepBeep is called the *engine*, which provides the basic classes for creating processors and functions, and contains a handful of general-purpose processors for manipulating traces. BeepBeep's remaining functionalities are dispersed across a number of *palettes*. In this chapter, we describe the basic processors and functions provided by BeepBeep's engine.

Function Objects

A **function** is something that accepts *arguments* and produces a return *value*. In BeepBeep, functions are "first-class citizens"; this means that every function that is to be applied on an event is itself an object, which inherits from a generic class called `Function`. For example, the negation of a Boolean value is a function object called `Negation`; the sum of two numbers is also a function object called `Addition`.

Function objects can be instantiated and manipulated directly. The BeepBeep classes `Booleans`, `Numbers` and `Sets` define multiple function objects to manipulate Boolean values, numbers and sets. These functions can be accessed through static member fields of these respective classes. Consider for example the following code snippet:

```
Function negation = Booleans.not;
Object[] out = new Object[1];
negation.evaluate(new Object[]{true}, out);
System.out.println("The return value of the function is: " + out[0]);
```

✎

The first instruction gets a reference to a `Function` object, corresponding to the static member field `not` of class `Booleans`. This field refers to an instance of a function called `Negation`. As a matter of fact, this is the only way to get an instance of `Negation`: its constructor is declared as `private`, which makes it impossible to create a new instance of the object using `new`. This is done on purpose, so that only one instance of `Negation` ever exists in a program –effectively making `Negation` a *singleton* object. We shall see that the vast majority of `Function` objects are singletons, and are referred to using a static member field of some other object.

In order to perform a computation, every function defines a method called `evaluate()`. This method takes two arguments; the first is an array of objects, corresponding to the input values of the function. The second is another array of objects, intended to receive the output values of the function. Hence, as for a processor, a function also has an input arity and an output arity.

For function `Negation`, both are equal to one: the negation takes one Boolean value as its argument, and returns the negation of that value. The second line of the example creates an array of size 1 to hold the return value of the function. Line 3 calls `evaluate`, with the Boolean value `true` used as the argument of the function. Finally, line 4 prints the result:

```
The return value of the function is: false
```

Functions with an input arity of size greater than 1 work in the same way. In the following example, we get an instance of the `Addition` function, and make a call on `evaluate` to get the value of 2+3.

```
Function addition = Numbers.addition;
addition.evaluate(new Object[]{2, 3}, out);
System.out.println("The return value of the function is: " + out[0]);
```

✎

As expected, the program prints:

```
The return value of the function is: 5.0
```

While the use of input/output arrays may appear cumbersome at first, it is mitigated by two things. First, you will seldom have to call `evaluate` on functions directly. Second, this mechanism makes it possible for functions to have arbitrary input and output arity; in particular, a function can have an output arity of 2 or more. Consider this last code example:

```
Function int_division = IntegerDivision.instance;
Object[] outs = new Object[2];
int_division.evaluate(new Object[]{14, 3}, outs);
System.out.println("14 divided by 3 equals " +
        outs[0] + " remainder " + outs[1]);
```

✎

The first instruction creates a new instance of another `Function` object, this time called `IntegerDivision`. From two numbers *x* and *y*, it outputs **two** numbers: the quotient and the remainder of the division of x by y. Note that contrary to the previous examples, this function was created by accessing the `instance` static field on class `IntegerDivision`. Most `Function` objects outside of utility classes such as `Booleans` or `Numbers` provide a reference to their singleton instance in this way. The remaining lines are again a call to `evaluate`: however, this time, the array receiving the output from the function is of size 2. The first element of the array is the quotient, the second is the remainder. Hence, the last line of the program prints this:

```
14 divided by 3 equals 4 remainder 2
```

Applying a Function on a Stream

A function is a "static" object: a call to `evaluate` receives a single set of arguments, computes a return value, and ends. In many cases, it may be desirable to apply a function to each event of a stream. In other words, we would like to "turn" a function into a processor applying this function. The processor responsible for this is called `ApplyFunction`. When instantiated, `ApplyFunction` must be given a `Function` object; it calls this function's `evaluate` on each input event, and returns the result on its output pipe.

In the following bit of code, an `ApplyFunction` is created by applying the Boolean negation function to an input trace of Boolean values:

```
QueueSource source = new QueueSource();
source.setEvents(false, true, true, false, true);
ApplyFunction not = new ApplyFunction(Booleans.not);
Connector.connect(source, not);
Pullable p = not.getPullableOutput();
for (int i = 0; i < 5; i++)
{
```

```
    boolean x = (Boolean) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

The first lines should be familiar at this point: they create a `QueueSource`, and give it a list of events to be fed upon request. In this case, we give the source a list of five Boolean values. In line 3, we create a new `ApplyFunction` processor, and give to its constructor the instance of the `Negation` function referred to by the static member field `Booleans.not`. Graphically, they can be represented as follows:

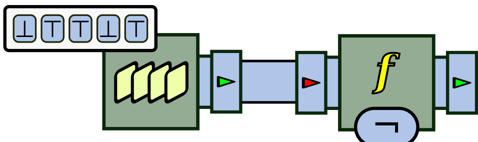


Figure 3.1: Applying a function on each input event transforms an input stream into a new output stream.

The `ApplyFunction` processor is represented by a box with a yellow *f* as its pictogram. This processor has an argument, the actual function it is asked to apply. By convention, function objects are represented by small rounded rectangles; the rectangle placed on the bottom side of the box represents the `Negation` function. Following the colour coding we introduced in the previous chapter, the stream manipulated is made of Boolean values; hence all pipes are painted in the blue-grey shade representing Booleans.

Calling `pull` on the `not` processor will return, as expected, the negation of the events given to the source. The program will print:

```
The event is: true
The event is: false
The event is: false
The event is: true
The event is: false
```

The input and output arity of the `ApplyFunction` matches that of the `Function` object given as its argument. Hence, a binary function will result in a binary processor. For instance, the following code example computes the pairwise addition of numbers from two streams:

```
QueueSource source1 = new QueueSource();
source1.setEvents(2, 7, 1, 8, 3);
QueueSource source2 = new QueueSource();
source2.setEvents(3, 1, 4, 1, 6);
ApplyFunction add = new ApplyFunction(Numbers.addition);
Connector.connect(source1, 0, add, 0);
Connector.connect(source2, 0, add, 1);
Pullable p = add.getPullableOutput();
for (int i = 0; i < 5; i++)
{
    float x = (Float) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

The reader may notice that this example is very similar to one we saw in the previous chapter. The difference lies in the fact that the original example used a special processor called `Adder` to perform the addition. Here, a generic `ApplyFunction` processor is used, to which the addition function is passed as a parameter. This difference is important: in the original case, there was no easy way to replace the addition by some other operation –apart from finding another purpose-built processor to do it. In the present case, changing the operation to some other binary function on numbers simply amounts to changing the function object given to `ApplyFunction`.

Function processors can be chained to perform more complex calculations, as is illustrated by the following code fragment:

```
QueueSource source1 = new QueueSource().setEvents(2, 7, 1, 8, 3);
QueueSource source2 = new QueueSource().setEvents(3, 1, 4, 1, 6);
QueueSource source3 = new QueueSource().setEvents(1, 1, 2, 3, 5);
ApplyFunction add = new ApplyFunction(Numbers.addition);
Connector.connect(source1, 0, add, 0);
Connector.connect(source2, 0, add, 1);
ApplyFunction mul = new ApplyFunction(Numbers.multiplication);
Connector.connect(source3, 0, mul, 0);
Connector.connect(add, 0, mul, 1);
Pullable p = mul.getPullableOutput();
for (int i = 0; i < 5; i++)
{
    float x = (Float) p.pull();
    System.out.println("The event is: " + x);
```

```
}
```

✎

Here, three sources of numbers are created; events from the first two are added, and the result is multiplied by the event at the corresponding position in the third stream. The diagram of such a program becomes more interesting:

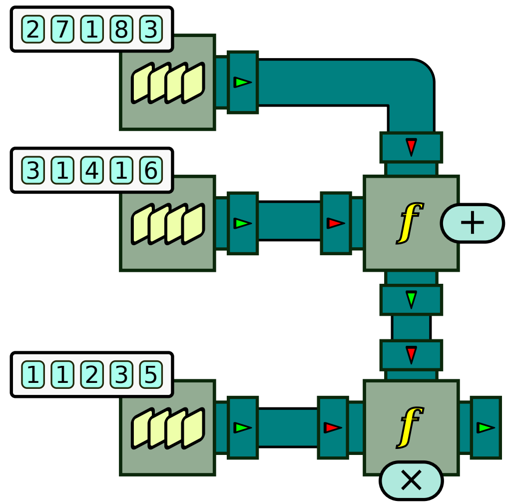


Figure 3.2: Chaining function processors.

The expected output of the program should look like this:

```
The event is: 5.0
The event is: 8.0
The event is: 10.0
The event is: 27.0
The event is: 45.0
```

Indeed, $(2+3)\times1=5$, $(7+1)\times1=8$, $(1+4)\times2=10$, and so on.

Function Trees

In the previous example, if the three input streams were named x, y and z, the processor chain created corresponds informally to the expression $(x+y)\times z$. However, having to write each arithmetical operator as an individual processor

can become tedious. After all, $(x+y)\times z$ is itself a function $f(x,y,z)$ of three variables; isn't there a way to create a `Function` object corresponding to this expression, and to give this expression to a single `ApplyFunction` processor?

Fortunately, the answer is yes. It is possible to create complex functions by composing simpler ones, through the use of a special `Function` object called the `FunctionTree`. As its name implies, a `FunctionTree` is effectively a tree structure whose nodes can either be:

- a `Function` object
- another `FunctionTree`
- a special type of variable, called a `StreamVariable`.

By nesting function trees within each other, it is possible to create complex expressions from simpler functions. As an example, let us revisit the previous program, and simplify the chain of `ApplyFunction` processors:

```
QueueSource source1 = new QueueSource().setEvents(2, 7, 1, 8, 3);
QueueSource source2 = new QueueSource().setEvents(3, 1, 4, 1, 6);
QueueSource source3 = new QueueSource().setEvents(1, 1, 2, 3, 5);
FunctionTree tree = new FunctionTree(Numbers.multiplication,
        new FunctionTree(Numbers.addition,
                StreamVariable.X, StreamVariable.Y),
        StreamVariable.Z);
ApplyFunction exp = new ApplyFunction(tree);
Connector.connect(source1, 0, exp, 0);
Connector.connect(source2, 0, exp, 1);
Connector.connect(source3, 0, exp, 2);
```

✎

We instantiate a new `FunctionTree` object after creating the three sources. The first argument is the function at the root of the tree; in an expression using parentheses, this corresponds to the operator that is to be evaluated *last* (here, the multiplication). The number of arguments that follow is variable: it corresponds to the expressions that are the arguments of the operator. In the example provided, the left-hand side of the multiplication is itself a `FunctionTree`. The operator of this inner tree is the addition, followed by its two arguments. Since we want to add the events coming from the first and second streams, these arguments are two `PullableException` objects. By convention, `StreamVariable.X` corresponds to input stream number 0, while `StreamVariable.Y` corresponds to input stream number 1. Finally, the right-hand side of the multiplication is `StreamVariable.Z`, which by convention

corresponds to input stream number 2.

This single-line instruction effectively created a new `Function` object with three arguments, which is then given to an `ApplyFunction` processor like any other function. Processor `exp` has an input arity of 3; all three sources can directly be connected into it: `source1` into input stream 0, `source2` into input stream 1, and `source3` into input stream 2. Graphically, this can be illustrated as follows:

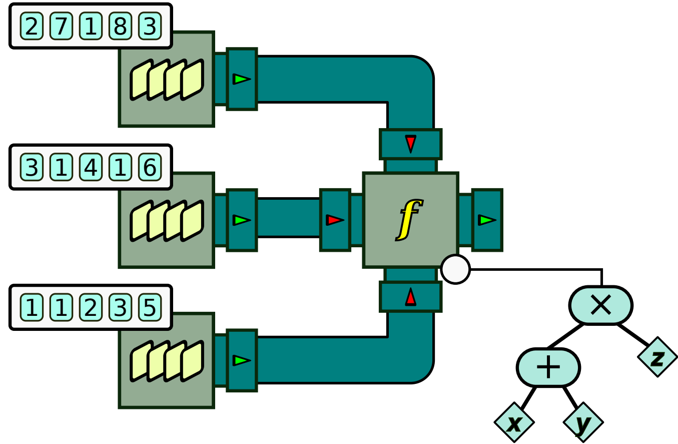


Figure 3.3: Chaining function processors.

As one can see, the single `ApplyFunction` processor is attached to a tree of functions, which corresponds to the object built by line 4. By convention, stream variables are represented by diamond shapes, with either the name of a stream variable (x, y or z), or equivalently with a number designating the input stream. Again, the colour of the nodes depicts the type of objects being manipulated. In the rest of the book and for the sake of clarity, the representation of a function as a tree will sometimes be forsaken; an inline notation such as $(x+y)\times z$ will be used to simplify the drawing.

Pulling events from `exp` will result in a similar pattern as before:

```
The event is: 5.0
The event is: 8.0
The event is: 10.0
The event is: 27.0
The event is: 45.0
```

Note that a stream variable may appear more than once in a function tree. Hence, an expression such as $(x+y)\times(x+z)$ is perfectly fine.

Forking a Stream

Sometimes, it may be useful to perform multiple separate computations over the same stream. In order to do so, one must be able to split the original stream into multiple identical copies. This is the purpose of the `Fork` processor.

As a first example, let us connect a queue source to create a fork processor that will replicate each input event in two output streams. This is the meaning of the number 2 passed as an argument to the fork's constructor.

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5);
Fork fork = new Fork(2);
Connector.connect(source, fork);
Pullable p0 = fork.getPullableOutput(0);
Pullable p1 = fork.getPullableOutput(1);
System.out.println("Output from p0: " + p0.pull());
System.out.println("Output from p1: " + p1.pull());
```

✎

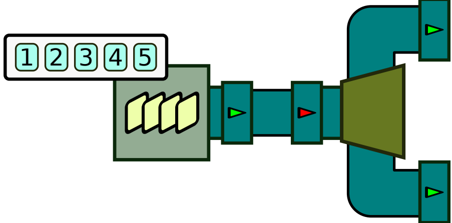


Figure 3.4: Pulling events from a fork.

We get Pullables on both outputs of the fork (`p0` and `p1`), and then pull a first event from `p0`. As expected, `p1` returns the first event of the source, which is the `Number` 1:

```
Output from p0: 1
```

We then pull an event from `p1`. Surprisingly enough, the output is:

```
Output from p1: 1
```

…and not 2 as might have been expected. This can be explained by the fact that each input event in the fork is replicated to all its output pipes. The fact that we pulled an event from `p0` has no effect on `p1`, and vice versa. The independence between the fork's two outputs is further illustrated by this sequence of calls:

```
System.out.println("Output from p0: " + p0.pull());
System.out.println("Output from p0: " + p0.pull());
System.out.println("Output from p1: " + p1.pull());
System.out.println("Output from p0: " + p0.pull());
System.out.println("Output from p1: " + p1.pull());
```

✎

Producing the output:

```
Output from p0: 2
Output from p0: 3
Output from p1: 2
Output from p0: 4
Output from p1: 3
```

Notice how each pullable moves through the input stream independently of calls to the other pullable.

Forks also exhibit a special behaviour in push mode. Consider the following example:

```
Fork fork = new Fork(3);
Print p0 = new Print().setSeparator("\n").setPrefix("P0 ");
Print p1 = new Print().setSeparator("\n").setPrefix("P1 ");
Print p2 = new Print().setSeparator("\n").setPrefix("P2 ");
Connector.connect(fork, 0, p0, INPUT);
Connector.connect(fork, 1, p1, INPUT);
Connector.connect(fork, 2, p2, INPUT);
Pushable p = fork.getPushableInput();
p.push("foo");
```

✎

We create a fork processor that will replicate each input event in three output streams. We now create three "print" processors. Each of them simply prints to the console whatever event they receive. Each of them is asked to append its printed line with a different prefix ("Px") to define who is printing what. Finally, we connect each of the three outputs streams of the fork (numbered 0,

1 and 2) to the input of each print processor. This corresponds to the following diagram:

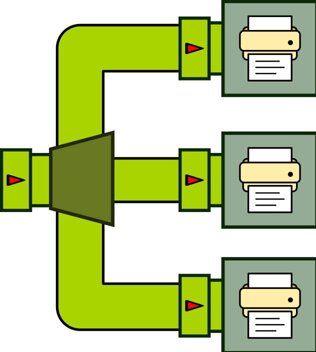


Figure 3.5: Pushing events into a fork.

Let's now push an event to the input of the fork and see what happens. We should see on the console:

```
P0 foo
P1 foo
P2 foo
```

The three lines should be printed almost instantaneously. This shows that all three print processors received their input event at the "same" time. This is not exactly true: the fork processor pushes the event to each of its outputs in sequence; however, since the time it takes to do so is so short, we can consider this to be instantaneous.

An important thing to keep in mind is that the fork, like almost all other BeepBeep processors, passes **references** to objects. In the previous example, the output events that are sent out are just three references to the same input event. This can cause bizarre side effects if the input event is a mutable object, and one of the downstream branches modifies that object. Consider a modified version of the previous example, as follows:

```
Fork fork = new Fork(3);
Print p0 = new Print().setSeparator("\n").setPrefix("P0 ");
Print p1 = new Print().setSeparator("\n").setPrefix("P1 ");
Print p2 = new Print().setSeparator("\n").setPrefix("P2 ");
Processor rf = new RemoveFirst();
Connector.connect(fork, 0, p0, INPUT);
```

```
Connector.connect(fork, 1, rf, INPUT);
Connector.connect(rf, p1);
Connector.connect(fork, 2, p2, INPUT);
```

✎

The difference lies in the fact that a special processor called `RemoveFirst` has been introduced between the fork's second output branch and the second `Print` processor. Let us suppose that this processor removes the first element of the list it receives and returns that list. This can be illustrated like this:

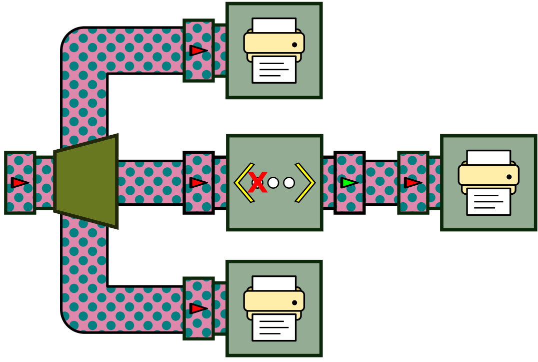


Figure 3.6: Pushing a mutable object into a fork and modifying that object in one of the downstream branches.

Let us now create a list and push it into the fork:

```
Pushable p = fork.getPushableInput();
List<Number> list = new ArrayList<Number>();
list.add(3); list.add(1); list.add(4);
p.push(list);
```

✎

The output of this program is:

```
P0 [3, 1, 4]
P1 [1, 4]
P2 [1, 4]
```

Notice how, this time, the `Print` processors do not all print the same thing. The input list `[3,1,4]` is first pushed into `p0`, which produces the first line of output. The list is then pushed into `rf`, which removes the first element of that list, and passes it to `p1`, which prints the second line of the output. The surprise

is on the third line of output, as we would expect p2 to receive the input list [3,1,4]. However, since elements are passed by reference, processor p2 is given a reference to the input list; it so happens that this list has been modified by rf just before, and is no longer in the same state as when it was pushed to the fork at the beginning of the program.

We do not recommend exploiting this side effect in your BeepBeep programs; although the fork seems to push events from top to bottom, the ordering is in fact undefined and should not be taken for granted. Most BeepBeep processors that are specific to mutable objects such as lists or sets take care of creating and returning a *copy* of the original object to avoid such unwanted behaviour (RemoveFirst is an exception, crafted only for this example). However, one must still be careful when passing around mutable objects that are referenced from multiple points in a program –as is the case in Java programming in general.

Cumulating Values

A variant of the function processor is the Cumulate processor. Contrary to all the processors we have seen so far, which are stateless, Cumulate is our first example of a **stateful** processor: this means that the output it returns for a given event depends on what it has output in the past. In other words, a stateful processor has a "memory", and the same input event may produce different outputs.

A Cumulate is given a function f of two arguments. Intuitively, if x is the previous value returned by the processor, its output on the next event y will be $f(x,y)$. Upon receiving the first event, since no previous value was ever set, the processor requires an initial value t to use in place of x.

As its name implies, Cumulate is intended to compute a cumulative "sum" of all the values received so far. The simplest example is when f is addition, and 0 is used as the start value t.

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5, 6);
Cumulate sum = new Cumulate(
        new CumulativeFunction<Number>(Numbers.addition));
Connector.connect(source, sum);
Pullable p = sum.getPullableOutput();
for (int i = 0; i < 5; i++)
```

```
{
    System.out.println("The event is: " + p.pull());
}
```

✎

We first wrap the `Addition` function into a `CumulativeFunction`. This object extends addition by defining a start value t. It is then given to the `Cumulate` processor. Graphically, this can be represented as follows:

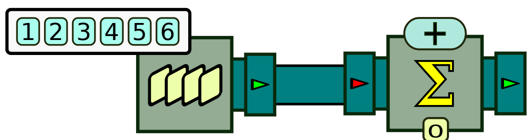


Figure 3.7: Computing the cumulative sum of numbers.

The `Cumulate` processor is represented by a box with the Greek letter sigma. On one side of the box is the function used for the cumulation (here addition), and on the other side is the start value t used when receiving the first event (here 0).

Upon receiving the first event y=1, the cumulate processor computes $f(x,1)$. Since no previous value x has yet been output, the processor uses the start value t=0 instead. Hence, the processor computes $f(0,1)$, that is, 0+1=1, and returns 1 as its first output event.

Upon receiving the second event y=2, the cumulate processor computes $f(x,2)$, with x being the event output at the previous step –in other words, x=1. This amounts to computing $f(1,2)$, that is 1+2=3. Upon receiving the third event y=3, the processor computes $f(3,3) = 3+3 = 6$. As can be seen, the processor outputs the cumulative sum of all values received so far:

```
The event is: 1.0
The event is: 3.0
The event is: 6.0
The event is: 10.0
...
```

Cumulative processors and function processors can be put together into a common pattern, illustrated by the diagram in Figure 3.8.

We first create a source of arbitrary numbers. The output of this processor is piped to a cumulative processor. Then, we create a source of 1s and sum it;

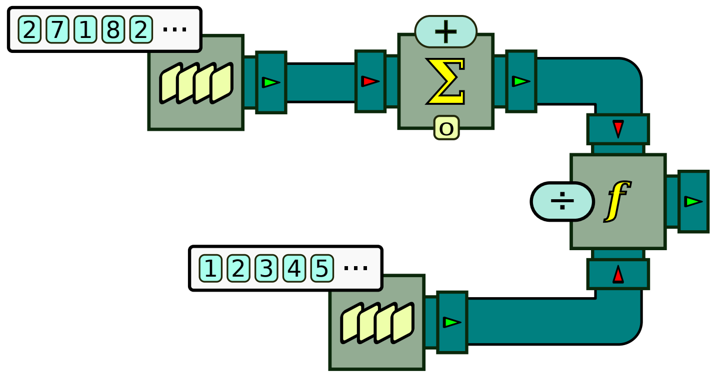


Figure 3.8: The running average of a stream of numbers.

this is done with the same process as above, but on a stream outputting the value 1 all the time. This effectively creates a counter outputting 1, 2, 3, etc. We finally divide one stream by the other.

Consider for example the stream of numbers 2, 7, 1, 8, etc. After reading the first event, the cumulative average is $2 \div 1 = 2$. After reading the second event, the average is $(2+7) \div (1+1)$, and after reading the third, the average is $(2+7+1) \div (1+1+1) = 3.33$ –and so on. The output is the average of all numbers seen so far. This is called the **running average**, and it occurs very often in stream processing. Coded, this corresponds to the following instructions:

```
QueueSource numbers = new QueueSource(1);
numbers.setEvents(new Object[]{2, 7, 1, 8, 2, 8, 1, 8, 2, 8,
        4, 5, 9, 0, 4, 5, 2, 3, 5, 3, 6, 0, 2, 8, 7});
Cumulate sum_proc = new Cumulate(
        new CumulativeFunction<Number>(Numbers.addition));
Connector.connect(numbers, OUTPUT, sum_proc, INPUT);
QueueSource counter = new QueueSource().setEvents(1, 2, 3, 4, 5, 6, 7);
ApplyFunction division = new ApplyFunction(Numbers.division);
Connector.connect(sum_proc, OUTPUT, division, LEFT);
Connector.connect(counter, OUTPUT, division, RIGHT);
```

✎

This example, however, requires a second queue just to count events received. Our chain of processors can be refined by creating a counter out of the original stream of values, as shown here:

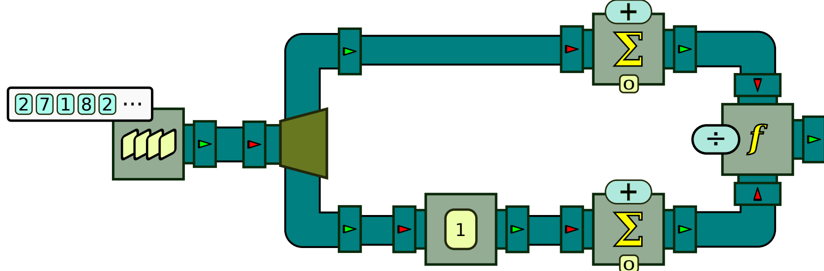


Figure 3.9: Running average not relying on an external counter.

We first fork the original stream of values in two copies. The topmost copy is used for the cumulative sum of values, as before. The bottom copy is sent into a processor called `TurnInto`; this processor replaces whatever input event it receives by the same predefined object. Here, it is instructed to turn every event into the number 1. This stream of 1s is then summed, effectively creating a counter that produces the stream 1, 2, 3, etc. The two streams are then divided as in the previous example.

It shall be noted that, `Cumulate` does not have to work only with addition, or even with numbers. Depending on the function *f*, cumulative processors can represent many other things. For example, in the next code snippet, a stream of Boolean values is created, and piped into a `Cumulate` processor, using logical conjunction ("and") as the function, and `true` as the start value:

```
QueueSource source = new QueueSource()
    .setEvents(true, true, false, true, true);
Cumulate and = new Cumulate(
        new CumulativeFunction<Boolean>(Booleans.and));
Connector.connect(source, and);
Pullable p = and.getPullableOutput();
for (int i = 0; i < 5; i++)
{
    System.out.println("The event is: " + p.pull());
}
```

✎

When receiving the first event (`true`), the processor computes its conjunction with the start value (also `true`), resulting in the first output event (`true`). The same thing happens for the second input event, resulting in the output

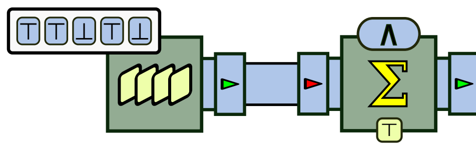


Figure 3.10: Using the Boolean "and" operator in a `Cumulate` processor.

event `true`. The third input event is `false`; its conjunction with the previous output event (`true`) results in `false`. From then on, the processor will return `false`, regardless of the input events. This is because the conjunction of `false` (the previous output event) with anything always returns `false`. Hence, the expected output of the program is this:

```
The event is: true
The event is: true
The event is: false
The event is: false
The event is: false
```

Intuitively, this processor performs the logical conjunction of all events received so far. This conjunction becomes false forever, as soon as a `false` event is received.

Trimming Events

Up until now, all the processors studied were **uniform**: for each input event, they emitted exactly one output event (or more precisely, for each input *front*, they emitted exactly one output *front*). Not all processors need to be uniform; as a first example, let us have a look at the `Trim` processor.

The purpose of `Trim` is simple: it discards a fixed number of events from the beginning of a stream. This number is specified by passing it to the processor's constructor. Consider for example the following code:

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5, 6);
Trim trim = new Trim(3);
Connector.connect(source, trim);
Pullable p = trim.getPullableOutput();
for (int i = 0; i < 6; i++)
{
```

```
    int x = (Integer) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

The `Trim` processor is connected to a source, and is instructed to trim 3 events from the beginning of the stream. Graphically, this is represented as follows:

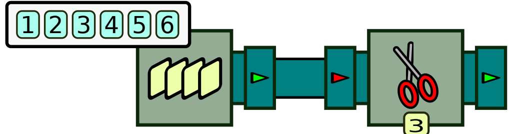


Figure 3.11: Pulling events from a `Trim` processor.

As one can see, the `Trim` processor is depicted as a box with a pair of scissors; the number of events to be trimmed is shown in a small box on one of the sides of the processor. Let us see what happens when we `pull` is called six times on `Trim`. The first call to `pull` produces the following line:

```
The event is: 4
```

This indeed corresponds to the *fourth* event in `source`'s list of events; the first three seem to have been cut off. But how can `trim` instruct `source` to start sending events at the fourth? In fact, the answer is that it does not. There is no way for a processor upstream or downstream to "talk" to another and give it instructions as to how to behave. What `trim` does is much easier: upon its first call to `pull`, it simply calls `pull` on its upstream processor four times, and discards the events returned by the first three calls.

At this point, `pull` behaves like `Passthrough`: it lets all events out without modification. The rest of the program goes as follows:

```
The event is: 5
The event is: 6
The event is: 1
The event is: 2
The event is: 3
```

Do not forget that a `QueueSource` loops through its list of events; this is why after reaching 6, it goes back to the beginning and outputs 1, 2 and 3.

The `Trim` processor behaves in a similar way in push mode, such as in this example:

```
Trim trim = new Trim(3);
Print print = new Print();
Connector.connect(trim, print);
Pushable p = trim.getPushableInput();
for (int i = 0; i < 6; i++)
{
    p.push(i);
}
```

✎

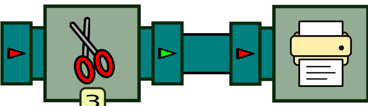


Figure 3.12: Pushing events into a `Trim` processor.

Here, we connect a `Trim` to a `Print` processor. The `for` loop pushes integers 0 to 5 into `trim`; however, the first three events are discarded, and do not reach `print`. It is only at the fourth event that a push on `trim` will result in a downstream push on `print`. Hence, the output of the program is:

```
3,4,5,
```

The `Trim` processor introduces an important point: from now on, the number of calls to `pull` or `push` is not necessarily equal across all processors of a chain. For example, in the last piece of code, we performed six `push` calls on `trim`, but `print` was pushed events only three times.

Coupled with `Fork`, the `Trim` processor can be useful to create two copies of a stream, offset by a fixed number of events. This makes it possible to output events whose value depends on multiple input events of the same stream. The following example (Figure 3.13) shows how a source of numbers is forked in two; on one of the copies, the first event is discarded. Both streams are then sent to a processor that performs an addition.

✎

On the first call on `pull`, the addition processor first calls `pull` on its first (top) input pipe, and receives from the source the number 1. The processor

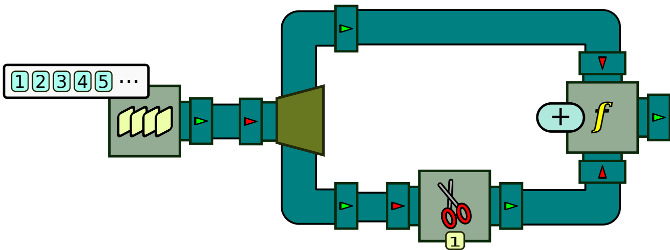


Figure 3.13: Computing the sum of two successive events.

then calls `pull` on its second (bottom) input pipe. Upon being pulled, the `Trim` processor calls `pull` on its input pipe *twice*: it discards the first event it receives from the fork (1), and returns the second (2). The first addition that is computed is hence 1+2=3, resulting in the output 3.

From this point on, the top and the bottom pipe of the addition processor are always offset by one event. When the top pipe receives 2, the bottom pipe receives 3, and so on. The end result is that the output stream is made of the sum of each successive pair of events: 1+2, 2+3, 3+4, etc. This type of computation is called a **sliding window**. Indeed, we repeat the same operation (here, addition) to a list of two events that progressively moves down the stream.

Sliding Windows

For a window of two events, like in the previous example, using a `Trim` processor may be sufficient. However, as soon as the window becomes larger, doing such a computation becomes very impractical (an exercise at the end of this chapter asks you to try with three events instead of two). The use of sliding windows is so prevalent in event stream processing that BeepBeep provides a processor that does just that. It is called, as you may guess, `Window`.

The `Window` processor is one of the two most complex processors in BeepBeep's core, and deserves some explanation. Suppose that we want to compute the sum of input events over a sliding window of width 3. That is, the first output event should be the sum of input events at positions 0 to 2; the second output event should be the sum of input events at positions 1 to 3, and so on. Each of these sequences of three events is called a **window**. The first step is to think

of a processor that performs the appropriate computation on each window, as if the events were fed one by one. In our case, the answer is easy: it is a `Cumulate` processor with addition as its function. If we pick any window of three successive events and feed them to a fresh instance of `Cumulate` one by one, the last event we collect is indeed the sum of all events in the window.

The second step is to encase this `Cumulate` processor within a `Window` processor, and to specify a window width (3 in our present case). A simple example of a window processor is the following piece of code:

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5, 6);
Cumulate sum = new Cumulate(
        new CumulativeFunction<Number>(Numbers.addition));
Window win = new Window(sum, 3);
Connector.connect(source, win);
Pullable p = win.getPullableOutput();
System.out.println("First window: " + p.pull());
System.out.println("Second window: " + p.pull());
System.out.println("Third window: " + p.pull());
```

✎

This code is relatively straightforward. The main novelty is the fact that the `Cumulate` processor, `sum`, is instantiated, and then given as a *parameter* to the `Window` constructor. As you can see, `sum` never appears in a call to `connect`. This is because the cumulative sum is what `Window` should compute internally on each window. Graphically, this is illustrated as follows:

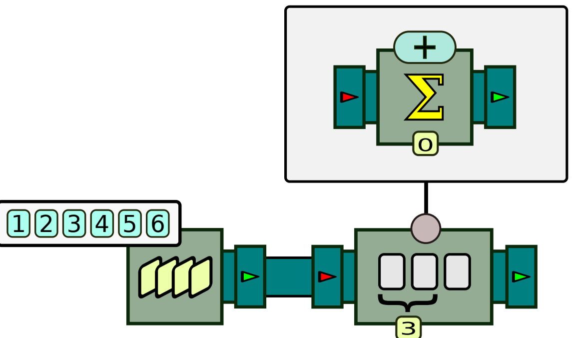


Figure 3.14: Using the `Window` processor to perform a computation over a sliding window of events.

The `Window` processor is depicted by a box with events grouped by a curly

bracket. The number under that bracket indicates the width of the window. On one side of the box is a circle leading to yet another box. This is to represent the fact that `Window` takes another processor as a parameter; in this box, we recognize the cumulative sum processor we used before. Notice how that processor lies alone in its box; as in the code fragment, it is not connected to anything. **Calling `pull` or `push` on that processor does not make sense, and will cause incorrect results, if not runtime exceptions.**

Let us now see what happens when we call `pull` on `win`. The window processor requires three events before being able to output anything. Since we just started the program, `win`'s window is currently empty. Therefore, three calls to `pull` are made on the source, in order to fetch the events 1, 2 and 3. Now that `win` has the correct number of input events, it pushes them into `sum` one by one. Since `sum` is a cumulative processor, it will successively output the events 1, 3 and 6 –corresponding to the sum of the first, the first two, and all three events, respectively. The window processor ignores all of these events except the last (6): this is the event that is returned from the first call to `pull`:

```
First window: 6.0
```

Things are slightly different on the second call to `pull`. This time, `win`'s window already contains three events; it only needs to discard the first event it received (1), and to let in one new event at the other end of the window. Therefore, it makes only one `pull` on `source`; this produces the event 4, and the contents of the window become 2, 3 and 4. As we can see, the window of three events has shifted one event forward, and now contains the second, third and fourth event of the input stream.

The window processor cannot push these three events to `sum` immediately. Remember that `sum` is a cumulative processor, and that it has already received three events. Pushing three more would not result in the sum of events in the current window. In fact, `sum` has a "memory", which must be wiped so that the processor returns to its original state. Every processor has a method allowing this, called `reset`. `Window` first calls `reset` on `sum`, and then proceeds to push the three events of the current window into it. The last collected event is 2+3+4=9, and hence the second line printed by the program is:

```
Second window: 9.0
```

The process then restarts for the third window, exactly in the same way as before. This results in the third printed line:

```
Third window: 12.0
```

Computing an average over a sliding window is a staple of event stream processing. This example pops up in every textbook on the topic, and virtually all event stream processing engines provide facilities to make such kinds of computations. However, typically, sliding windows only apply to streams of numerical values, and the computation over each window is almost always one of a few *aggregation* functions, such as `min`, `max`, `avg` (average) or `sum`. BeepBeep distinguishes itself from most other tools in that `Window` computations are much more generic. Basically, **any computation can be encased in a sliding window**. To prove our point, consider the following chain of processors:

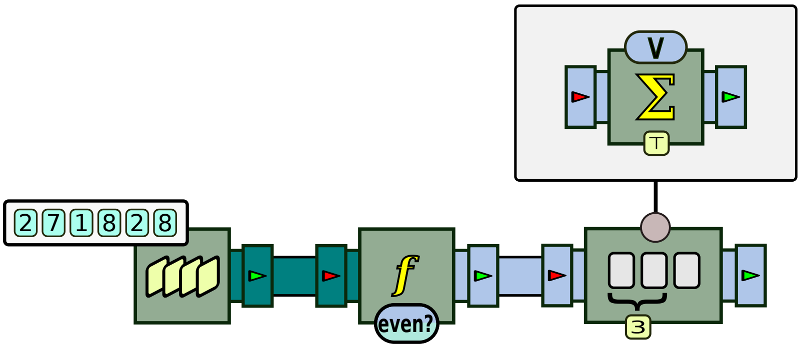


Figure 3.15: Sliding windows can be applied on streams that are not numeric.

✎

A numerical stream is passed into an `ApplyFunction` processor; the function evaluates whether a number is even, using a built-in function called `IsEven`. This function takes a number as input, and returns a Boolean value. This stream of *Booleans* is then piped into a `Window` processor, which will handle windows of Booleans. On each window, a `Cumulate` processor computes the disjunction (logical "or") of all events in the window. On a given window of three successive events, the output is `true` if and only if there is at least one even number. The end result of this whole chain is a stream of Booleans; it returns `false` whenever three input events in a row are odd, and `true` otherwise.

As we can see, although this example makes use of a `Window` processor, its meaning is far from the numerical aggregation functions used in classical

event stream processing systems. As a matter of fact, BeepBeep's very general way of handling windows is unique among existing stream processors.

This example also marks the first time we have a chain of processors where multiple event types are mixed. The first end of the chain manipulates numbers (green pipes), while the last part of the chain has Boolean events (grey-blue). Notice how function `IsEven` in the diagram has two colours. The bottom part represents the input (green, for numbers), while the top part represents the output (grey-blue, for Booleans). Similarly, the input pipe of the `ApplyFunction` processor is green, while its output pipe is grey-blue, for the same reason.

Grouping Processors

We claimed a few moments ago that "anything can be encased in a sliding window". This means that, instead of a single processor, we could give `Window` a more complex chain, like the one that computes the running average of a stream of numbers, as illustrated below.

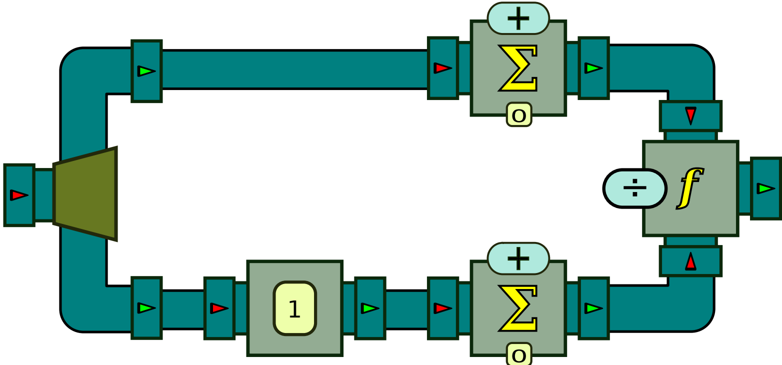


Figure 3.16: A chain of processors computing the running average of a stream.

But how exactly can we give this *chain* of processors as a parameter to `Window`? Its constructor expects a *single* `Processor` object, so which one shall we give? If we pass the input fork, how is `Window` supposed to know where the output of the chain is? And conversely, if we pass the downstream processor that computes the division, how is `Window` supposed to learn where to push events?

The answer to this is a special type of processor called `GroupProcessor`. The `GroupProcessor` allows a user to encapsulate a complete chain of processors into a composite object which can be manipulated as if it were a single `Processor`. In other words, `GroupProcessor` hides its contents into a "black box", and only exposes the input and output pipes at the very ends of the chain.

Let us revisit a previous example (✎), and use a group processor, as in the following code fragment.

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5, 6);
GroupProcessor group = new GroupProcessor(1, 1);
{
    Fork fork = new Fork(2);
    ApplyFunction add = new ApplyFunction(Numbers.addition);
    Connector.connect(fork, 0, add, 0);
    Trim trim = new Trim(1);
    Connector.connect(fork, 1, trim, 0);
    Connector.connect(trim, 0, add, 1);
    group.addProcessors(fork, trim, add);
    group.associateInput(0, fork, 0);
    group.associateOutput(0, add, 0);
}
Connector.connect(source, group);
Pullable p = group.getPullableOutput();
for (int i = 0; i < 6; i++)
{
    float x = (Float) p.pull();
    System.out.println("The event is: " + x);
}
```

✎

After creating a source of numbers, we create a new empty `GroupProcessor`. The constructor takes two arguments, corresponding to the input and output arity of the group. Here, our group processor will have one input pipe, and one output pipe. The block of instructions enclosed inside the pair of braces put contents inside the group. The first six lines work as usual: we create a fork, a trim and a function processor, and connect them all together. The remaining three lines are specific to the creation of a group. The seventh line calls method `addProcessors()`; this puts the created processors inside the group object.

However, merely putting processors inside a group is not sufficient. The

GroupProcessor has no way to know what are the inputs and outputs of the chain. This is done with calls to associateInput() and associateOutput(). The eighth line tells the group processor that its input pipe number 0 should be connected to input pipe number 0 of fork. The ninth line tells the group processor that its output pipe number 0 should be connected to output pipe number 0 of add.

It is now possible to use group as if it were a single processor box. The remaining lines connect source to group, and fetch a Pullable object from group's output pipe. Graphically, this is illustrated as follows:

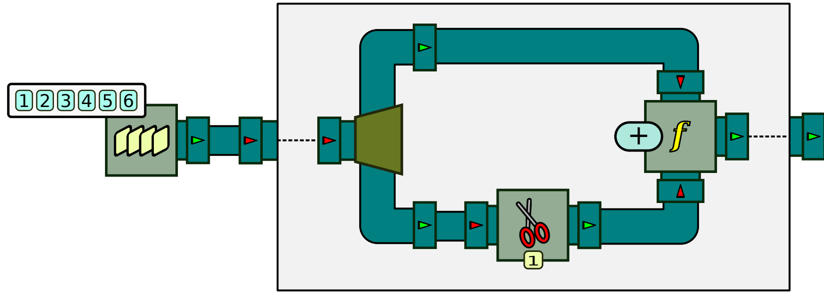


Figure 3.17: Simple usage of a GroupProcessor.

Note how the chain of processors is enclosed in a large rectangle, which has one input and one output pipe. The calls to associateInput() and associateOutput() correspond to the dashed lines that link the group's input pipe to the input pipe of the enclosed chain, and similarly for the output pipe.

Equipped with a GroupProcessor, it now becomes easy to compute the average over a sliding window we started this section with. This can be illustrated as in Figure 3.18.

The code corresponding to this picture is shown below:

```
QueueSource numbers = new QueueSource(1);
numbers.setEvents(new Object[]{2, 7, 1, 8, 2, 8, 1, 8, 2, 8,
        4, 5, 9, 0, 4, 5, 2, 3, 5, 3, 6, 0, 2, 8, 7});
GroupProcessor group = new GroupProcessor(1, 1);
{
    Fork fork = new Fork(2);
    Cumulate sum_proc = new Cumulate(
            new CumulativeFunction<Number>(Numbers.addition));
    Connector.connect(fork, TOP, sum_proc, INPUT);
```

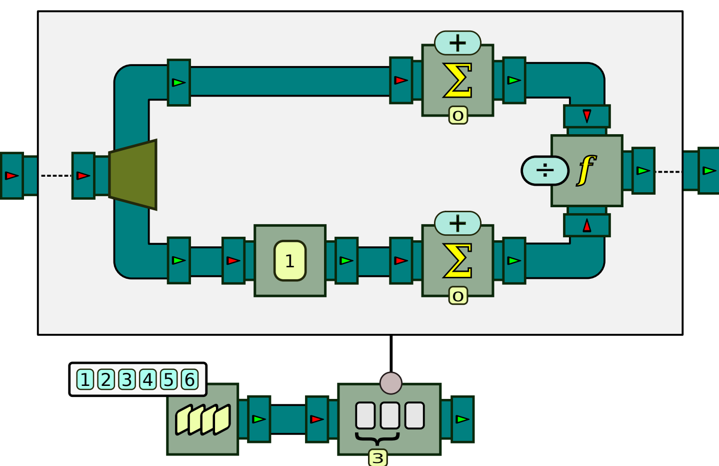


Figure 3.18: Computing the running average over a sliding window.

```
    TurnInto ones = new TurnInto(1);
    Connector.connect(fork, BOTTOM, ones, INPUT);
    Cumulate counter = new Cumulate(
            new CumulativeFunction<Number>(Numbers.addition));
    Connector.connect(ones, OUTPUT, counter, INPUT);
    ApplyFunction division = new ApplyFunction(Numbers.division);
    Connector.connect(sum_proc, OUTPUT, division, LEFT);
    Connector.connect(counter, OUTPUT, division, RIGHT);
    group.addProcessors(fork, sum_proc, ones, counter, division);
    group.associateInput(0, fork, 0);
    group.associateOutput(0, division, 0);
}
Window win = new Window(group, 3);
Connector.connect(numbers, win);
```

✎

Groups can have an arbitrary input and output arity, as is shown in Figure 3.19.

Here, we create two copies of the input stream offset by one event. These two streams are sent to an `ApplyFunction` processor that evaluates function `IntegerDivision`, which we encountered earlier in this chapter. This function has an input and output arity of 2. We want the group processor to output both the quotient and the remainder of the division as two output streams.

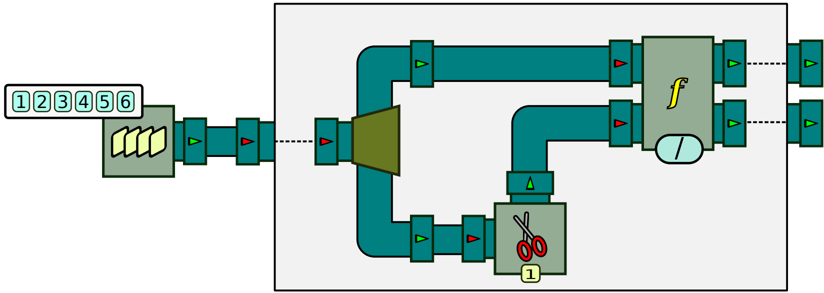


Figure 3.19: A group processor with more than one output pipe.

Since the group has two output pipes, two calls to `associateOutput` must be made. The first associates output 0 of the function processor to output 0 of the group, and the second associates output 1 of the function processor to output 1 of the group. The code creating the group is hence written as follows:

```
Fork fork = new Fork(2);
ApplyFunction div = new ApplyFunction(IntegerDivision.instance);
Connector.connect(fork, 0, div, 0);
Trim trim = new Trim(1);
Connector.connect(fork, 1, trim, 0);
Connector.connect(trim, 0, div, 1);
group.addProcessors(fork, trim, div);
group.associateInput(0, fork, 0);
group.associateOutput(0, div, 0);
group.associateOutput(1, div, 1);
```

✎

Decimating Events

A common task in event stream processing is to discard events from an input stream at periodic intervals. This process is called **decimation**. The two common ways to decimate events are:

- based on a fixed number of events (*count decimation*), and
- based on a fixed interval of time (*time decimation*).

In this section, we concentrate on the former. To perform count decimation, BeepBeep provides a processor called `CountDecimate`. Let us push events to

such a processor, as in the following code fragment.

```
CountDecimate dec = new CountDecimate(3);
Print print = new Print();
Connector.connect(dec, print);
Pushable p = dec.getPushableInput();
for (int i = 0; i < 10; i++)
{
    p.push(i);
}
```

✎

Here, a `CountDecimate` processor is created and connected into a `Print` processor. The decimate processor is instructed to keep one event for every 3, and to discard the others. This is the meaning of value 3 passed to its constructor, which is called the *decimation interval*, as shown in the following:

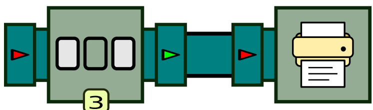


Figure 3.20: Pushing events to a `CountDecimate` processor.

The `CountDecimate` processor is designated by a pictogram in which some events are transparent, representing decimation. Like many other processors receiving parameters, the decimation interval is written on one side of the box. Let us now push the integers 0 to 9 into this processor, and watch the output printed at the console. The result is the following:

```
0,3,6,9,
```

As expected, the processor passed the first event (0), discarded the next two (1 and 2), then passed the fourth (3), and so on.

An important point must be made when `CountDecimate` is used in pull mode, as in the chain shown in Figure 3.21.

In such a case, the events received by each call to `pull` will be 1, 4, 7, etc. That is, after outputting event 1, the decimate processor does not ignore our next two calls to `pull` by returning nothing. Rather, it pulls three events from the queue source and discards the first two.

The decimate processor can be mixed with the other processors seen so far.

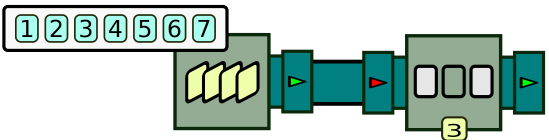


Figure 3.21: Pulling events from a `CountDecimate` processor.

For example, we have seen earlier how we can use a `Window` processor to calculate the sum of events on a sliding window of width n. We can affix a `CountDecimate` processor to the end of such a chain to create what is called a **hopping window**. Contrary to sliding windows, where the content of two successive windows overlap, hopping windows are disjoint. For example, one can compute the sum of the first five events, then the sum of the next five, and so on. The difference between the two types of windows is illustrated in the following figure; sliding windows are shown at the left, and hopping windows are shown at the right.

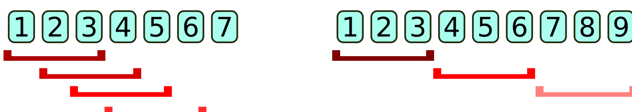


Figure 3.22: Difference between a sliding window (left) and a hopping window (right).

As one can see, hopping windows can be created out of sliding windows of width n by simply keeping one window out of every n.

Filtering Events

The `CountDecimate` processor acts as a kind of filter, based on the events' position. If an input event is at a position that is an integer multiple of the decimation interval, it is sent in the output; otherwise, it is discarded. Apart from the `Trim` processor we have encountered earlier, this is so far the only way to discard events from an input stream.

The `Filter` processor allows a user to keep or discard events from an input stream in a completely arbitrary way. In its simplest form, a `Filter` has two input pipes and one output pipe. The first input pipe is called the *data pipe*:

it consists of the stream of events that needs to be filtered. The second input pipe is called the *control pipe*: it receives a stream of Boolean values. As its name implies, this Boolean stream is responsible for deciding what events coming into the data pipe will be kept, and what events will be discarded. The event at position n in the data stream is sent to the output, if and only if the event at position n in the control stream is the Boolean value `true`.

As a first example, consider the following piece of code, which connects two sources to a `Filter` processor:

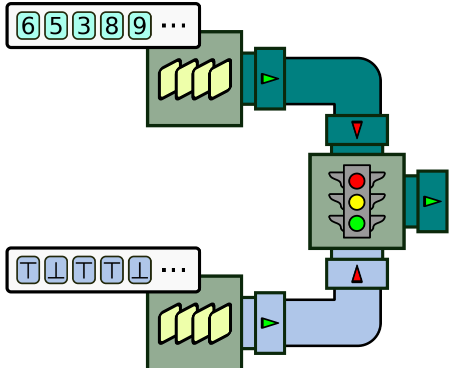


Figure 3.23: Filtering events.

The first source corresponds to the data stream, and in this case consists of a sequence of arbitrary numbers. The second source corresponds to the control stream, which we populate with randomly chosen Boolean values. These two sources are connected to a `Filter`. By convention, the *last* input pipe of a filter is the control stream; the remaining input pipes are the data streams. It is a common mistake to connect what is intended to be the control stream into the wrong pipe of the filter. This is illustrated below:

```
QueueSource source_values = new QueueSource();
source_values.setEvents(6, 5, 3, 8, 9, 2, 1, 7, 4);
QueueSource source_bool = new QueueSource();
source_bool.setEvents(true, false, true, true,
        false, false, true, false, true);
Filter filter = new Filter();
connect(source_values, OUTPUT, filter, TOP);
connect(source_bool, OUTPUT, filter, BOTTOM);
Pullable p = filter.getPullableOutput();
for (int i = 0; i < 5; i++)
```

```
{
    int x = (Integer) p.pull();
    System.out.printf("Output event #%d is %d\n", i, x);
}
```

✎

The `Filter` is represented by a box with a traffic light as a pictogram. Since the data stream is made of numbers, both the data input pipe and the output pipes are coloured in green. Obviously, the control pipe, which is made of Booleans, is always grey-blue.

The last part of the program, as usual, simply pulls on the output of the `Filter` and prints what is received. In this case, the output of the program is:

```
Output event #0 is 6
Output event #1 is 3
Output event #2 is 8
Output event #3 is 1
Output event #4 is 4
```

As we can see, the events from `source_values` that are output are only those at a position where the corresponding value in `source_bool` is `true`. At position 0, the event in `source_bool` is `true`, so the value 6 is output. On the second call to `pull`, `filter` pulls on both its input pipes; it receives the value 5 from `source_values`, and the value `false` from `source_bool`. Since the control pipe holds the value `false`, the number 5 has to be discarded, meaning that `filter` has nothing to output. Consequently, it pulls again on its input pipes to receive another event front. This time, it receives the pair 3/`true`, so it can return 3 as its second event.

Since the output of events depends entirely on the contents of the control stream, the relative positions of the events in the input and output streams do not follow any predictable pattern:

- Event at position 0 in the output corresponds to event at position 0 in the input;
- Event at position 1 in the output corresponds to event at position 2 in the input;
- Event at position 2 in the output corresponds to event at position 3 in the input;
- Event at position 3 in the output corresponds to event at position 7 in the input.

Note also that on a call to `pull`, a filter *must* return something. Therefore, it will keep pulling on its input pipes until it receives an event front where the control event is `true`. If that event never comes, **the call to `pull` will never end**. As a small exercise, try to replace all the Boolean values in `source_bool` by `false`, and run the program again. You will see that nothing is printed on the console, and that the program loops forever.

Like other processors in BeepBeep, the filtering mechanism is very generic and flexible. Any stream can be filtered, as long as a control stream is provided. As we have seen in our example, this control stream does not even need to be related to the data stream: any Boolean stream will do. In many cases, though, the decision on whether to filter an event or not depends on the event itself. For example, we would like to keep an event only if it is an even number. How can we accommodate such a situation?

The solution is to combine the `Filter` with another processor we have seen earlier, the `Fork`. From a given input stream, we use a fork to create two copies. The first copy is our data stream, and is sent directly to the filter's data pipe. We then use the second copy of the stream to evaluate a condition that will serve as our data stream. This is exactly what is done in the following example:

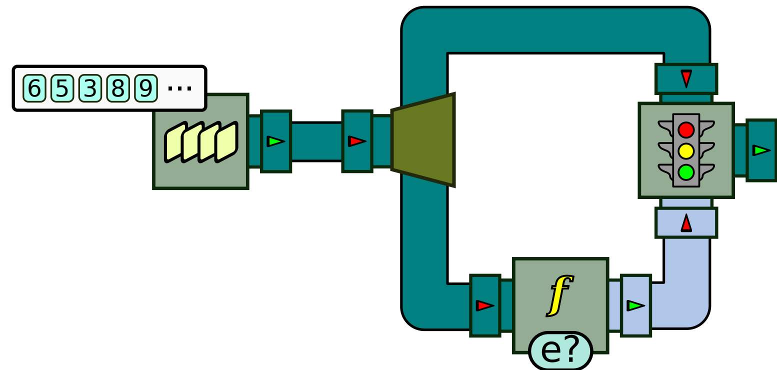


Figure 3.24: Filtering events.

```
QueueSource source_values = new QueueSource();
source_values.setEvents(6, 5, 3, 8, 9, 2, 1, 7, 4);
Fork fork = new Fork(2);
connect(source_values, fork);
Filter filter = new Filter();
connect(fork, LEFT, filter, LEFT);
ApplyFunction condition = new ApplyFunction(Numbers.isEven);
```

```java
connect(fork, RIGHT, condition, INPUT);
connect(condition, OUTPUT, filter, RIGHT);
Pullable p = filter.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    int x = (Integer) p.pull();
    System.out.printf("Output event #%d is %d\n", i, x);
}
```

✎

As we can see, the bottom part of the chain passes the input stream through an `ApplyFunction` processor, which evaluates the function `IsEven`. This function turns the stream of numbers into a stream of Booleans, which is then connected to the filter's control pipe. The end result of this chain is to produce an output stream where all odd numbers from the input stream have been removed. Obviously, if a more complex condition needs to be evaluated, a `FunctionTree` can be used instead of a single function. As a matter of fact, users are not limited to a single `ApplyFunction` processor, and can create whatever chain of processors they wish, as long as it produces a Boolean stream!

Slicing a Stream

The `Filter` is a powerful processor in our toolbox. Using a filter, we can take a larger stream and create a "sub-stream" –that is, a stream that contains a subset of the events of the original stream. Using forks, we can even create *multiple* different sub-streams from the same input stream. For example, we can separate a stream of numbers into a sub-stream of even numbers on one side, and a sub-stream of odd numbers on the other. This is perfectly possible, as Figure 3.25 shows.

However, we can see that this drawing contains lots of repetitions. The chains of processors at both ends of the first fork are almost identical; the only difference is the function passed to each instance of `ApplyFunction`: in the top chain, even numbers are kept, while in the bottom chain, a negation is added to the condition, so that odd numbers are kept. The two output pipes at the far right of the diagram hence produce a stream of even numbers (at the top) and a stream of odd numbers (at the bottom).

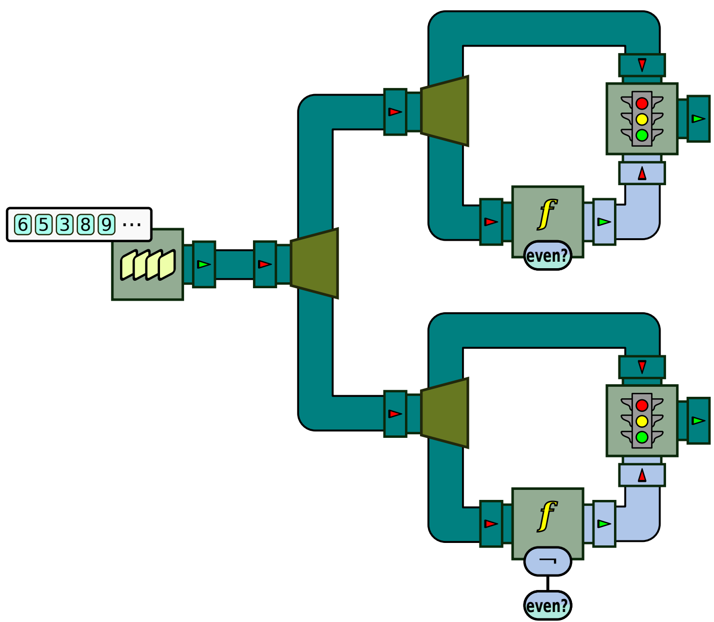


Figure 3.25: Creating two sub-streams of events: a stream of odd numbers, and a stream of even numbers.

Suppose, however, that we need to perform further processing on both these sub-streams. For example, we would like to compute their cumulative sum. We would need to repeat the same chain of processors at the end of both pipes. Suppose further that we would like to create *three* sub-streams instead of two, by filtering events according to their value modulo 3 (which returns either 0, 1 or 2): we would then need to copy-paste even more processors and pipes. There should be a better way to proceed.

Fortunately, there is. In fact, there are many situations in which we would like to separate a stream into multiple sub-streams, and perform the same computation over each of these sub-streams separately. Because this situation is a recurrent one, BeepBeep provides `Slice`, a processor dedicated to this specific task.

Creating a `Slice` processor works in a similar way to `Window`. Two parameters are needed to construct `Slice`:

1. The first is a **slicing function**, which is evaluated on each incoming event. The value of that function determines to which sub-stream that

event belongs. Typically, there will exist as many sub-streams as there are possible output values for the slicing function. These sub-streams are called *slices*, hence the name of the processor.
2. The second is a **slice processor**. A different instance of this processor is created for each possible value of the slicing function. When an incoming event is evaluated by the slicing function, it is then pushed to the instance of the slice processor associated to that value.

As with the `Window` processor, the `Slice` processor expects a single object as its slice processor. To pass a chain of multiple processors, it must be encapsulated into a `GroupProcessor`, as seen previously.

To illustrate the operation of a slice processor, consider the following code example:

```
QueueSource source = new QueueSource();
source.setEvents(1, 6, 4, 3, 2, 1, 9);
Function slicing_fct = new IdentityFunction(1);
GroupProcessor counter = new GroupProcessor(1, 1);
{
    TurnInto to_one = new TurnInto(new Constant(1));
    Cumulate sum = new Cumulate(
        new CumulativeFunction<Number>(Numbers.addition));
    Connector.connect(to_one, sum);
    counter.addProcessors(to_one, sum);
    counter.associateInput(INPUT, to_one, INPUT);
    counter.associateOutput(OUTPUT, sum, OUTPUT);
}
Slice slicer = new Slice(slicing_fct, counter);
Connector.connect(source, slicer);
Pullable p = slicer.getPullableOutput();
for (int i = 0; i < 10; i++)
{
    Object o = p.pull();
    System.out.println(o);
}
```

✎

In this program, we first create a simple source of numbers, and connect it to an instance of `Slice`. In this case, the slicing function is the `IdentityFunction`: this function returns its input as is. The slice processor is a simple counter that increments every time an event is received, which we encapsu-

late into a `GroupProcessor`. Since there will be one such counter instance for each different input event, the slicer effectively keeps count of how many times each value has been seen in its input stream. Graphically, this can be represented as:

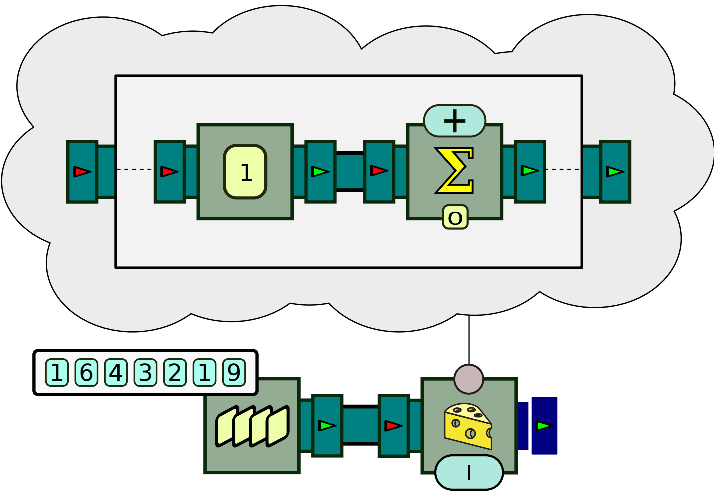


Figure 3.26: Using a `Slice` processor.

The `Slice` processor is represented by a box with a piece of cheese (yes, cheese) as its pictogram. Like the `Window` processor, one of its arguments (the slicing function) is placed on one side of the box, and the other argument (the slice processor) is linked to the box by a circle and a line. We took the liberty of putting the slice processor inside a "cloud" instead of a plain rectangle. As expected, this slice processor is itself a group that encapsulates a `TurnInto` and a `Cumulate` processor.

Let us now see what happens when we start pulling events on `slicer`. On the first call to `pull`, `slicer` pulls on the source and receives the number 1. It evaluates the slicing function, which (obviously) returns 1. It then seeks into its memory for an instance of the slice processor associated to the value 1. Since there is none, `slicer` creates a new copy of the slice processor, and pushes the value 1 into it. It then collects the output from that slice processor, which is (again) the value 1.

The last step is to return something to the call to `pull`. What a slicer outputs is always a Java `Map` object. The keys of that map correspond to values of the slicing function, and the value for each key is the last event produced

by the corresponding slice processor. Every time an event is received, the slicer returns as its output the newly updated map. At the beginning of the program, the map is empty; this first call to `pull` will add a new entry to the map, associating the value 1 to the slice "1". The first line printed by the program is the contents of the map, namely:

```
{1=1.0}
```

The second call to `pull` works in a similar fashion. The slicer receives the value 6 from the source; no slice processor exists for that value, so a new one is created. Event 6 is pushed into it, and the output value (1) is collected. A new entry is added to the map, associating slice 6 to the value 1. Note that the previous entry is still there, so that the next printed line is:

```
{1=1.0, 6=1.0}
```

A similar process occurs for the next three input events, creating three new map entries:

```
{1=1.0, 4=1.0, 6=1.0}
{1=1.0, 3=1.0, 4=1.0, 6=1.0}
{1=1.0, 2=1.0, 3=1.0, 4=1.0, 6=1.0}
```

Something slightly different happens in the next call to `pull`. The `slicer` receives the number 1, evaluates the slice function, which returns 1. It turns out that this is a value for which a slice processor already exists. Therefore, `slicer` retrieves that processor instance, and pushes the value 1 into it. Note that for this slice processor, this is the *second* time it is given an event; since it acts as a counter, it returns the value 2. Then, `slicer` updates its map by associating the value 2 to slice 1, which replaces the original entry. The map that is returned on the call to `pull` is:

```
{1=2.0, 2=1.0, 3=1.0, 4=1.0, 6=1.0}
```

The end result of this processor chain is that it keeps track of how many times each number has been seen in the input stream so far.

As we can see, each copy of the slice processor is fed the sub-trace of all events for which the slicing function returns the same value. Different results can be obtained by using a different slicing function. Let us go back to our original example, where we would like to create sub-streams of odd and even numbers, and to compute their cumulative sum separately. This time, the slicing function will determine if a number is odd or even; this task can be done using the function `IsEven`. Passing it to the `Slice` processor will generate two streams:

one comprising the numbers for which `IsEven` returns `true` (the even numbers), and another comprising the numbers for which `IsEven` returns `false` (the odd numbers). We then affix as the slice processor a `GroupProcessor` that encapsulates a chain computing the cumulative sum of numbers.

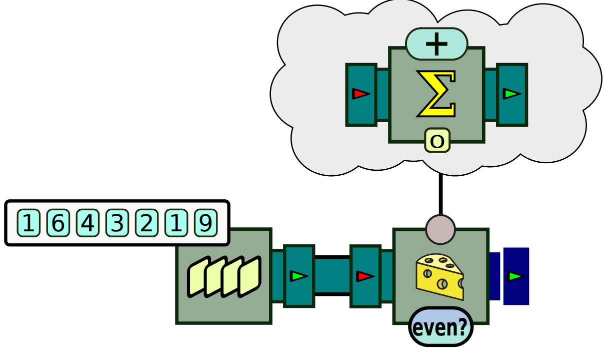


Figure 3.27: Adding odd and even numbers separately.

The end result of this program is a map with two keys (`true` and `false`), associated with the cumulative sum of even numbers and odd numbers, respectively.

Keeping the Last Event

In some cases, it may be desirable to take an action only when all the events from an input source have been consumed. For example, one may want to decimate a stream by keeping one event every 100, but still output the last event if the stream has, say, 560 events. When writing a processor chain that produces a plot from an input file, it can be useful to first read and process all the file, before triggering the generation of the plot; this would bring a better performance than producing a new plot upon every input event. In BeepBeep, a few processors have functionalities allowing users to deal with the "last" event of a stream.

The first is called `KeepLast`. As its name implies, its task is to discard every event received from upstream, and to output only the last. This can be illustrated by the following program:

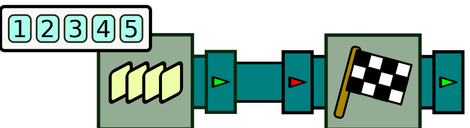


Figure 3.28: Keeping the last event.

A QueueSource is connected to the KeepLast processor, represented by a box with a checkered flag. In code, this corresponds to the following program:

```
QueueSource src = new QueueSource().setEvents(1, 2, 3, 4, 5);
src.loop(false);
KeepLast kl = new KeepLast();
Connector.connect(src, kl);
Pullable p = kl.getPullableOutput();
while (p.hasNext())
{
    Object o = p.next();
    System.out.print(o);
}
```

✎

Notice how the source is instructed *not* to loop through its list of events. This means that after outputting the number 5, any subsequent calls to hasNext will return false. The program then enters a loop, and pulls events from the output of the KeepLast processor until none is available. Running this program produces the single number 5, which, indeed, is the last event produced by the upstream source src.

Once the KeepLast processor has output the last event received from upstream, it does not return any other event. Subsequent calls to hasNext on kl will all return false, which means that the loop in the program is executed only once. Conversely, KeepLast will keep pulling on its upstream processor until it receives the indication that the last event has been produced. This means that on a processor chain that has "no end", such as a QueueSource that loops through its list of events forever, the call to hasNext on KeepLast will never return.

In pull mode, Identifying the last event can easily be done, precisely by looking at the return value of hasNext when pulling on an upstream processor. The situation is less obvious in push mode, such as in the following diagram:

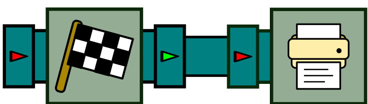


Figure 3.29: Pushing events on the `KeepLast` processor.

How can a processor push an event, and indicate that this is the last? This is illustrated by the following program:

```
QueueSource src = new QueueSource().setEvents(1, 2, 3, 4, 5);
src.loop(false);
KeepLast kl = new KeepLast();
Connector.connect(src, kl);
Pullable p = kl.getPullableOutput();
while (p.hasNext())
{
    Object o = p.next();
    System.out.print(o);
}
```

✎

Here, events are repeatedly pushed to the `KeepLast` processor. The first three calls to `push` have no noticeable effect: the `Print` processor does not print anything. Only the last line of the program will trigger the printing of an event. As a matter of fact, the `Pushable` interface defines a method called `notifyEndOfTrace`. Calling this method is the way of telling the underlying processor: “the last event I pushed was the last event of the stream”. In the case of `KeepLast`, this triggers a call to `push` on its downstream processor, containing the last event that was received. Obviously, it makes no sense to call `notifyEndOfTrace`, and push more events afterwards. As a matter of fact, the behaviour of a processor in such a situation is undefined, and it is not recommended doing so.

Not all processors react to a call to `notifyEndOfTrace`. For example, `ApplyFunction` does nothing special when reaching the end of an input stream. However, the `CountDecimate` processor *can* be told to output the last event of a stream, regardless of whether it is placed at an integer multiple of the decimation interval. To this end, it suffices to pass the Boolean value `true` as a second argument to `CountDecimate`’s constructor. To illustrate this, we revisit an earlier example using `CountDecimate` in the following program.

```
CountDecimate dec = new CountDecimate(3, true);
Print print = new Print();
Connector.connect(dec, print);
Pushable p = dec.getPushableInput();
for (int i = 0; i < 8; i++)
{
    p.push(i);
}
p.notifyEndOfTrace();
```

✎

The difference lies in the fact that `CountDecimate` has been instantiated with `true`; the processor behaves normally when calling `push`, and sends to the `Print` processor every third input event. However, the call to `notifyEndOfTrace` triggers the output of the last event. Therefore, the program prints at the console:

```
0,3,6,7,
```

Notice how number 7 should not have been output under normal circumstances.

In this chapter, we have covered the dozen or so fundamental processors provided by BeepBeep's core. These processors allow us to manipulate event streams in various ways: applying a function to each event, filtering, decimating, slicing and creating sliding windows. Most of these processors are "type agnostic": the actual type of events they handle has no influence in the way they operate. Therefore, a large number of event-processing tasks can be achieved by appropriately combining these basic building blocks together. We could show many other examples of graphs combining processors in various ways; these were rather left as exercises in the section below. A little time is required to get used to decomposing a problem in terms of streams; this is why we recommend that you try some of these exercises and develop your intuition before moving on to the next chapter.

Exercises

1. Write a processor chain that computes the sum of each event with the one two positions away in the stream. That is, output event 0 is the sum of input events 0 and 2; output event 1 is the sum of input events 1 and 3, and so on. You can do this using a very slight modification to one of the examples in this chapter.

2. Using `CountDecimate` and `Trim`, write a processor chain that outputs events at position $3n+1$ and discards the others. That is, from the input stream, the output should contain events at position 1, 4, 7, 10, etc.

3. Using only the `Fork`, `Trim` and `ApplyFunction` processors, write a processor chain that computes the sum of all three successive events. (Hint: you will need two `Trims`.)

4. Write a processor chain that outputs events at position n^2. That is, from the input stream, the output should contain events at position 1, 4, 9, 16, etc.

5. Write a processor chain that computes the Fibonacci sequence. The sequence starts with numbers 1 and 1; every subsequent number is the sum of the previous two.

6. Write a processor chain receiving a stream of numerical values, and which flattens to zero any input value that lies below a predefined threshold k. The chain should leave the values greater than k as they are. (Hint: use a function called `IfThenElse`.)

7. Write a `GroupProcessor` that takes a stream of numbers, and alternates their sign: it multiplies the first event by -1, the second by 1, the third by -1, and so on. This processor only needs to work in pull mode.

8. The value of pi can be estimated using the Leibniz formula. According to this formula, pi is four times the infinite expression 1/1 - 1/3 + 1/5 - 1/7 + 1/9… Create a chain of processors that produces an increasingly precise approximation of the value of pi using this formula.

9. Write a processor chain that computes the running variance of a stream of numbers. The variance can be calculated by the expression $E[X^2]-E[X]^2$, where $E[X]$ is the running average, and $E[X^2]$ is the running average of the square of each input event.

10. Write a processor chain that takes as input a stream of numbers, and outputs a stream of Booleans. Output event at position i should be true if and only if input event at position i is more than two standard deviations away from the running average of the stream at this point. (Hint: the standard deviation is the square root of the running variance.)

11. Write a processor chain that prints “This is a multiple of 5” when a multiple of 5 is pushed, and prints “This is something else” otherwise.

12. From a stream of Boolean values, write a processor chain that computes the number of times a window of width 3 contains more `false` than `true`. That is, from the input stream TTFFTFTT, the processor should output the values 0, 1, 2, 3, 3, 3.

13. Write a processor chain that counts the number of times a positive number is immediately followed by a negative number.

4 Advanced Features

The previous chapters have shown the fundamental concepts around BeepBeep and the basic processors that can be used in general use cases. In this chapter, we shall see a number of more special-purpose processors comprised in BeepBeep's core likely to be used in one of your processor chains.

Lists, sets and maps

Up to this point, all the examples we have seen use event streams that are one of Java's primitive types: numbers (`int`s or `float`s), `String`s and `Boolean`s. However, we mentioned at the very beginning that one of BeepBeep's design principles is that everything (that is, any Java object) can be used as an event. To this end, the `util` package provides functions and processors to manipulate a few common data structures, especially lists, sets and maps.

A few of these functions are grouped under the `Bags` utility class. It contains references to functions that can be used to query arbitrary collections of objects.

`Bags.getSize` refers to a function `GetSize` that takes a Java `Collection` object for input, and returns the size of this collection. For example, if `list` is a `List` object containing a few elements, one could use `GetSize` like any other function:

```
Object[] outs = new Object[1];
Bags.getSize.evaluate(new Object[]{list}, outs);
// outs[0] contains the size of list
```

`Bags.contains` refers to a function `Contains` that takes as input a Java `Collection` and an object *o*, and returns a Boolean value indicating whether

the collection contains *o*. Its usage can be illustrated in the following code example:

```
QueueSource src1 = new QueueSource();
src1.addEvent(UtilityMethods.createList(1f, 3f, 5f));
src1.addEvent(UtilityMethods.createList(4f, 2f));
src1.addEvent(UtilityMethods.createList(4f, 4f, 8f));
src1.addEvent(UtilityMethods.createList(6f, 4f));
QueueSource src2 = new QueueSource();
src2.setEvents(1);
ApplyFunction contains = new ApplyFunction(Bags.contains);
Connector.connect(src1, 0, contains, 0);
Cumulate counter = new Cumulate(
    new CumulativeFunction<Number>(Numbers.addition));
Connector.connect(src2, counter);
Connector.connect(counter, 0, contains, 1);
Pullable p = contains.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    System.out.println(p.pull());
}
```

✎

First, a `QueueSource` is created, as usual; note that this time, each event in the source is itself a *list* (method `createList` is a small utility method that creates a `List` object out of its arguments). Then, this source is piped as the first argument of an `ApplyFunction` processor that evaluates `Bags.contains`; its second argument comes from a stream of numbers that increments by one. The end result is a stream where the n-th output event is the value `true` if and only if the n-th input list in `src1` contains the value n. This is illustrated by Figure 4.1.

This drawing introduces the "polka dot" pattern. The base colour to represent collections (sets, lists or arrays) is pink; the dots on the pipes are used to indicate the type of the elements inside the collection (here, numbers). When the type of the elements inside the collection is unknown or varies, the pipes will be represented in flat pink without the dots. Note also the symbol used to depict the `Contains` function.

As expected, the output of the program is:

```
true
```

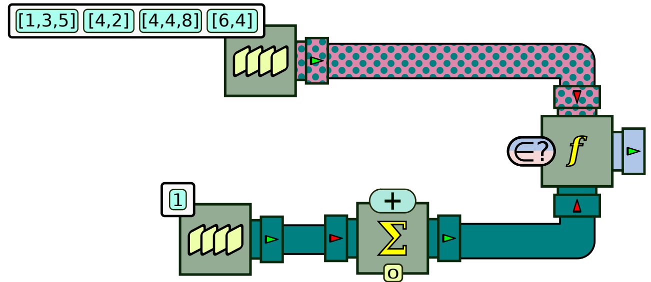


Figure 4.1: A first event stream with a more complex data structure.

```
true
false
true
```

The `Bags` class also provides a function called `ApplyToAll`. This function is instantiated by giving it a `Function` object *f*; given a set/list/array, `ApplyToAll` returns a *new* set/list/array whose content is the result of applying *f* to each element. This can be shown in the following example:

```
List<Object> list = UtilityMethods.createList(-3, 6, -1, -2);
Object[] out = new Object[1];
Function f = new Bags.ApplyToAll(Numbers.absoluteValue);
f.evaluate(new Object[]{list}, out);
System.out.println(out[0]);
```

✎

The output of this code snippet is indeed a new list with the absolute value of the elements of the input list:

```
[3.0, 6.0, 1.0, 2.0]
```

The `FilterElements` function can be used to remove elements from a collection. Like `ApplyToAll`, `FilterElements` is instantiated by passing a `Function` object *f* to its constructor. This function must be 1:1 and must return a Boolean value. Given a set/list/array, `FilterElements` will return a new set/list/array containing only elements for which *f* returns `true`. Using the same list as above, the following code:

```
Function filter = new Bags.FilterElements(Numbers.isEven);
filter.evaluate(new Object[]{list}, out);
System.out.println(out[0]);
```

✎

will produce this output:

```
[6, -2]
```

It is also possible to take the input of multiple streams, and to create a collection out of each front of events. This can be done with the help of function `ToList`. Consider the following code example:

```
QueueSource src1 = new QueueSource().setEvents(3, 1, 4, 1, 6);
QueueSource src2 = new QueueSource().setEvents(2, 7, 1, 8);
QueueSource src3 = new QueueSource().setEvents(1, 1, 2, 3, 5);
ApplyFunction to_list = new ApplyFunction(
        new Bags.ToList(Number.class, Number.class, Number.class));
Connector.connect(src1, 0, to_list, 0);
Connector.connect(src2, 0, to_list, 1);
Connector.connect(src3, 0, to_list, 2);
Pullable p = to_list.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    System.out.println(p.pull());
}
```

✎

We first create three sources of numbers, and pipe them into an `ApplyFunction` processor that is given the `ToList` function. When instantiated, this function must be given the type (that is, the `Class` object) of each of its inputs. Here, the function is instructed to receive three arguments, and is told that all three are instances of `Number`.

Graphically, this can be illustrated as in Figure 4.2 (note the symbol used to represent `ToList`):

When run, this program will take each front of events from the sources, and create a list object of size three with those three events. The output of this program is therefore:

```
[3, 2, 1]
[1, 7, 1]
[4, 1, 2]
[1, 8, 3]
```

The functions `ToSet` and `ToArray` operate in a similar way, but respectively

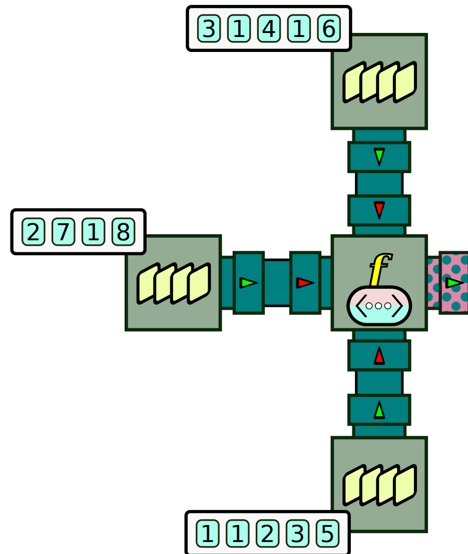


Figure 4.2: Creating lists from the input of multiple streams.

create a Set object and an array instead of a list.

Finally, the Bags class also defines a Processor object called RunOn. When instantiated, A must be given a 1:1 processor P. When it receives a collection as its input, RunOn takes each element of the collection, pushes it into P, and collects its last output.

Consider the following code example:

```
QueueSource src1 = new QueueSource();
src1.addEvent(UtilityMethods.createList(1f, 3f, 5f));
src1.addEvent(UtilityMethods.createList(4f, 2f));
src1.addEvent(UtilityMethods.createList(4f, 4f, 8f));
src1.addEvent(UtilityMethods.createList(6f, 4f));
Bags.RunOn run = new Bags.RunOn(new Cumulate(
    new CumulativeFunction<Number>(Numbers.addition)));
Connector.connect(src1, run);
Pullable p = run.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    System.out.println(p.pull());
}
```

✎

A RunOn processor is created, and is given a Cumulate processor that is instructed to compute the cumulative sum of a stream of events. When receiving a collection, RunOn pushes each element into a fresh copy of Cumulate; the last event is collected and returned. The end result is a program that computes the sum of elements in each set:

```
9.0
6.0
16.0
10.0
```

The following picture shows how to depict the RunOn processor graphically. Like the other processors seen earlier (such as Window and Slice), RunOn can take any Processor object as an argument. However, if we want to pass a chain of processors, we must carefully encapsulate that chain inside a GroupProcessor.

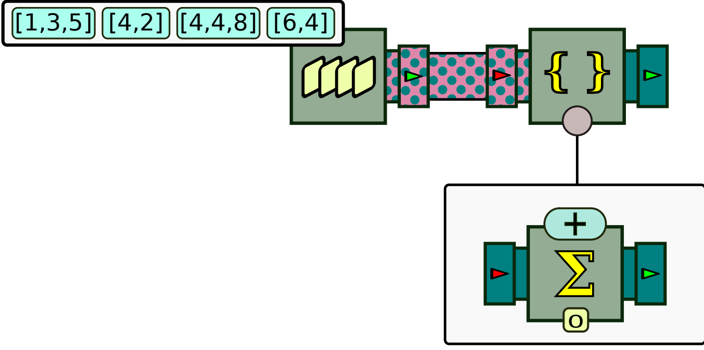


Figure 4.3: Applying a processor on collections of events with RunOn.

SET-SPECIFIC OBJECTS

The util package also provides a few functions and processors specific to some particular types of collections. The Sets class has a member field Sets.isSubsetOrEqual which refers to a function IsSubsetOrEqual that compares two Set objects. It also defines a processor PutInto which receives arbitrary objects as input, and accumulates them into a set, which it returns as its output.

The following program shows the basic usage of `PutInto`.

```
QueueSource src = new QueueSource().setEvents("A", "B", "C", "D");
Sets.PutInto put = new Sets.PutInto();
Connector.connect(src, put);
Pullable p = put.getPullableOutput();
Set<Object> set1, set2;
p.pull();
set1 = (Set<Object>) p.pull();
System.out.println("Set 1: " + set1);
p.pull();
set2 = (Set<Object>) p.pull();
System.out.println("Set 2: " + set2);
System.out.println("Set 1: " + set2);
```

✎

It produces the following output:

```
Set 1: [A, B]
Set 2: [A, B, C, D]
Set 1: [A, B, C, D]
```

Note how after the second call to `pull`, the variable `set1` is a set that contains the first two events, "A" and "B". Two calls to `pull` later, variable `set2` contains, as expected, the first four events. The last call to `println` is more surprising. It reveals that `set1` now also contains the first four events! This is because the variables `set1` and `set2` are actually two references to the same object. In other words, processor `PutInto` keeps returning the same `Set`, each time with a new element added to it. We say that `PutInto` is a **mutator** processor: it modifies the state of the objects it returns.

If we want to have a different set for every output event, we must rather use `PutIntoNew`. Upon each input event, this processor creates a new set, copies the content of the previous one, and adds the new event into it. Since this processor performs a copy every time, it runs much slower than `PutInto`.

LIST-SPECIFIC OBJECTS

Functions and processors that work on arbitrary collections obviously also work on lists. BeepBeep provides a few more of these for collections that are *ordered*, such as lists and arrays. For example, `NthElement` is a function that returns the element at the n-th position in an ordered collection.

The `Lists` class defines two processors that work on lists in a special way. The first is called `Pack` and has two input pipes. The first, called the *data* pipe, is a stream of arbitrary events. The second, called the *control* pipe, is a stream of Boolean values. You may remember that the `Filter` processor seen in the previous chapter had two similarly-named input pipes.

Processor `Pack` accumulates events received from the input pipe, as long as the corresponding event in the control pipe is the Boolean value `false`. When the value in the control pipe is `true`, `Pack` outputs the list of events accumulated so far, instantiates a new empty list, and puts the incoming event into it. Consider the following example:

```
QueueSource src1 = new QueueSource();
src1.setEvents(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5);
QueueSource src2 = new QueueSource();
src2.setEvents(false, true, false, false, false, true, false, true);
Lists.Pack pack = new Lists.Pack();
Connector.connect(src1, 0, pack, 0);
Connector.connect(src2, 0, pack, 1);
Pullable p = pack.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    System.out.println(p.pull());
}
```

✎

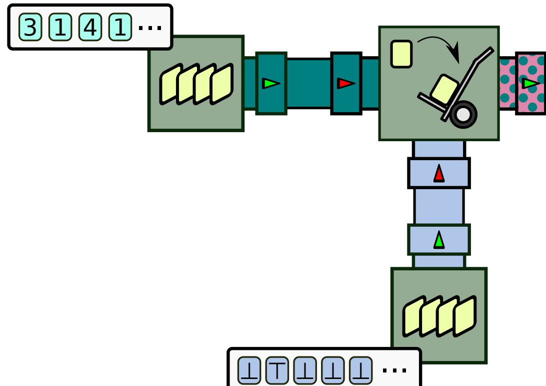


Figure 4.4: Packing events into lists using the `Pack` processor.

We create a data and a control stream, connect them to a `Pack` processor and

pull events from its output. The program prints:

```
[3]
[1, 4, 1, 5]
[9, 2]
[6, 5]
```

One can see how the control stream acts as a "trigger" telling the `Pack` processor when to release a list of events.

The `Unpack` processor is the exact opposite of `Pack`. It receives a stream of lists, and outputs the event of each list one by one:

```
QueueSource src1 = new QueueSource();
src1.addEvent(UtilityMethods.createList(1f, 3f, 5f));
src1.addEvent(UtilityMethods.createList(4f, 2f));
src1.addEvent(UtilityMethods.createList(4f, 4f, 8f));
src1.addEvent(UtilityMethods.createList(6f, 4f));
Lists.Unpack unpack = new Lists.Unpack();
Connector.connect(src1, 0, unpack, 0);
Pullable p = unpack.getPullableOutput();
for (int i = 0; i < 6; i++)
{
    System.out.println(p.pull());
}
```

✎

Figure 4.5: Unpacking events from a stream of lists using the `Unpack` processor.

Of course, putting an `Unpack` processor next to a `Pack` processor will recreate the original stream. This means that the following processor chain is equivalent to a `Passthrough` processor on the stream of numbers 3, 1, 4, …

Note that the end result of this program does not depend on the Boolean stream at the bottom. This control stream merely changes the way events are grouped into lists, but does not change the relative ordering of each event. Since the lists are unpacked immediately, the output from `Unpack` will always be the same.

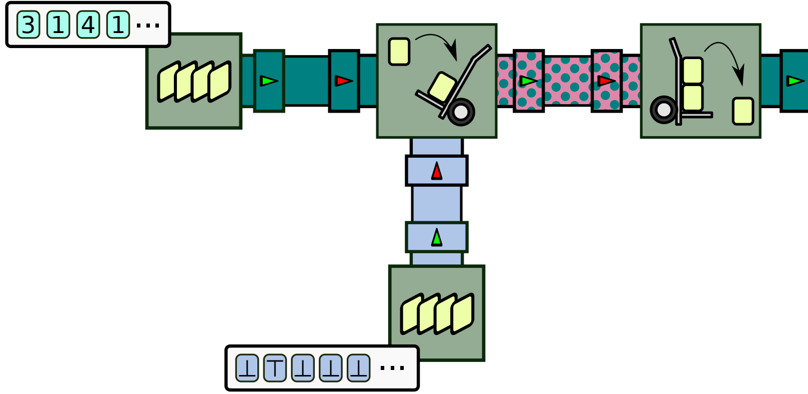


Figure 4.6: Chaining a `Pack` and an `Unpack` processor.

One final remark must be made about `Unpack` when it is used in push mode. Consider the following simple chain:

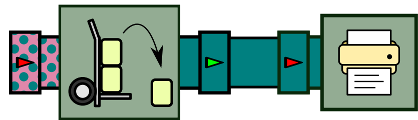


Figure 4.7: Using the `Unpack` processor in push mode.

Suppose that `p` is `Unpack`'s `Pushable` object; what do you suppose the following program will print?

```
List<Object> list = UtilityMethods.createList(1, 2, 3, 4);
System.out.println("Before first push");
p.push(list);
list = UtilityMethods.createList(5, 6, 7);
System.out.println("\nBefore second push");
p.push(list);
System.out.println("\nAfter second push");
```

✎

Let us see what happens on the first call to `push`. The `Unpack` processor is given a list of numbers. Its task is to output these numbers one by one; since we are in push mode, these numbers will hence be pushed to its output pipe and down to the `Print` processor. But how many such numbers will be pushed? Only the first one?

The answer is **all at once**. That is, on the single call to `push`, the `Unpack` processor will push all four events one after the other. The output of the program is therefore:

```
Before first push
1,2,3,4,
Before second push
5,6,7,
After second push
```

Notice how the first call to `push` on `Unpack` results in four calls to `push` on the downstream `Print` processor. We had already seen that the number of calls to `pull` may not be uniform across a processor chain; we now know that the same is true for calls to `push`. As a matter of fact, the `Pack` and `Unpack` processors are called **non-uniform**: for a single output event, the number of output events they produce is not always the same. In contrast, uniform processors produce the same number of output events for each input event.

We have already seen other non-uniform processors before: the `Filter`, `CountDecimate` and `Trim` processors sometimes produce zero output event from an input event. Here, we see a non-uniform processor in the opposite way: it sometimes produces more than one output event from a single input event.

MAP-SPECIFIC OBJECTS

There is one last Java collection we haven't talked about: `Map`. As you know, a map is a data structure associating arbitrary *keys* to arbitrary *values*. A map can be queried for the value corresponding to a key using a method called `get()`. BeepBeep provides a `Maps` class that defines a few functions and processors specific to the manipulation of such maps. The first one is `Get`, which, as you may guess, fetches a value from a map given the name of a key. A simple usage would be the following:

```
Map map = ...
Function get = new Maps.Get("foo");
Object[] out = new Object[1];
get.evaluate(new Object[]{map}, out);
// out[0] contains the value of key foo
```

The `Maps` class also defines a processor maps that works in the same way as the one we have seen in `Sets` and `Lists`. It receives two input streams: the first

one is made of the “keys”, and the second one is made of the “values”. When receiving an event front, it creates a key-value pair from the two events and uses it to update the map, which it then returns. For example, the following program:

```
QueueSource keys = new QueueSource()
    .setEvents("foo", "bar", "foo", "baz");
QueueSource values = new QueueSource()
    .setEvents(1, "abc", "def", 6);
Maps.PutInto put = new Maps.PutInto();
Connector.connect(keys, 0, put, 0);
Connector.connect(values, 0, put, 1);
Pullable p = put.getPullableOutput();
for (int i = 0; i < 4; i++)
{
    System.out.println(p.pull());
}
```

✎

. . . will produce the following output:

```
{foo=1}
{bar=abc, foo=1}
{bar=abc, foo=def}
{bar=abc, foo=def, baz=6}
```

Note how the map is *updated*: if a key already exists in the map, its corresponding value is replaced by the new one. Also note how types can be mixed in the map: the value for key “foo” is first a number, and is replaced later by string. A variant of `PutInto` is called `ArrayPutInto`, which takes a single input stream, whose events are *arrays*. The first element of the array contains the key, and the second contains the value.

One last function of interest is called `Values`. This function takes a map as input, and returns the collection made of all the values occurring in the key-value pairs it contains. This function performs the equivalent of the `values()` method in Java’s `Map` interface.

Pumps and Tanks

All the processor chains provided as examples operate either in pull mode or in push mode. In pull mode, a chain must be closed upstream by having the chain start by processors of input arity 0. The opposite applies for push mode: the chain must be closed downstream by ending each branch by a processor of output arity 0. Because of this, we cannot combine pull and push in the same chain.

There exist situations, however, where it would be desirable to use both modes. Consider this simple program:

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4);
Print print = new Print();
Pullable pl = source.getPullableOutput();
Pushable ps = print.getPushableInput();
while (true)
{
    ps.push(pl.pull());
    Thread.sleep(1000);
}
```

✎

Here, events are pulled from a `QueueSource`, which are then pushed into the `Print` processor to display them on the console. Since the only way for a processor to push events downstream is to be pushed events from upstream, `source` cannot be asked to push events to `print`. For the reverse reason, `print` cannot be asked to pull events from `source`. The only way to make these two objects interact is by the hand-written `while` loop, which acts as a “bridge” between an upstream chain working in pull mode, and a downstream chain working in push mode.

While the loop works well in this example, the fact that the link between `print` and `source` is done outside of BeepBeep’s objects has a few negative implications:

- The upstream and downstream parts are two separate groups of processors that are completely unaware of each other. Since they are not a continuous chain of processors, they cannot be encapsulated in a `GroupProcessor` and passed to other processors. This also breaks the

traceability chain, meaning that it becomes impossible to trace an output event back to one or more input events.
- The connection between `source` and `print` is not done through `Connector`'s `connect` method; this bypasses a few sanity checks, such as the verification of input and output types compatibility.
- There is no easy way to start/stop this process upon request, or to ask this chain to process one event at a time. The control flow of the program must stay in the `while` loop as long as events need to be processed.

Fortunately, BeepBeep has a processor that can do the equivalent of our manual pull/push loop. This processor is appropriately called the `Pump`. Using a `Pump`, the previous program can be replaced by this one:

```
QueueSource source = new QueueSource().setEvents(1, 2, 3, 4);
Pump pump = new Pump(1000);
Print print = new Print();
Connector.connect(source, pump, print);
Thread th = new Thread(pump);
th.start();
```

✎

A pump is created and connected between `source` and `print`. This object is then placed inside a Java `Thread`, and this thread is then started. This has for effect of starting the pump itself, which will push/pull one event every 1,000 milliseconds (as was specified in its constructor). As you can understand, a pump implements Java's `Runnable` interface so that it can be put inside a thread. Graphically, this program can be represented as follows:

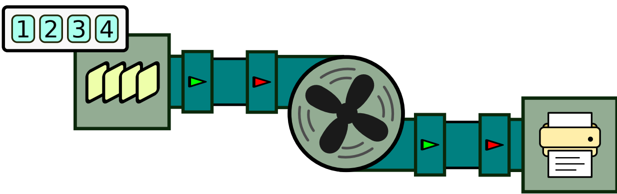


Figure 4.8: A chain of processors using a `Pump`.

Notice that, contrary to the examples seen so far, this chain of processors is closed on both ends. The only way to move events around is by the internal action of the pump. This can also be done manually by calling method `turn`, which performs a single pull/push.

There also exists a processor that performs the reverse operation, by bridging an upstream "push" section to a downstream "pull" section. This processor is called the `Tank`. In a processor chain that uses a tank, events are pushed from upstream until they reach the tank, at which point they are accumulated indefinitely. The downstream part of the chain can be queried using calls to `pull`; these calls propagate until they reach the tank, outputting the accumulated events one by one.

Consider the following program:

```
ApplyFunction to_number = new ApplyFunction(Numbers.numberCast);
Tank tank = new Tank();
Cumulate sum = new Cumulate(
    new CumulativeFunction<Number>(Numbers.addition));
Connector.connect(to_number, tank, sum);
```

✎

A function processor casting strings into numbers is connected to a tank, which itself is connected to a processor that computes a cumulative sum. This is illustrated by Figure 4.1.

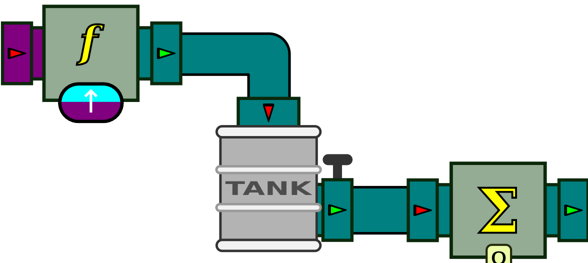


Figure 4.9: A chain of processors using a `Tank`.

A telling sign that `Tank` is the true dual of `Pump`: note that the chain, this time, is open at both ends. This means that events can be pushed from one end, and pullled from the other independently.

```
Pushable ps = to_number.getPushableInput();
Pullable pl = sum.getPullableOutput();
ps.push("1");
ps.push("2");
System.out.println(pl.pull());
System.out.println(pl.pull());
```

```
System.out.println(pl.pull());
```

✎

However, events cannot be pulled from the tank more than were pushed to it beforehand. On the last call to `pull` in our example, the tank is empty; this will throw an exception, as if the processor were connected to nothing. Therefore, the program outputs:

```
1.0
3.0
Exception in thread "main" java.util.NoSuchElementException
```

Basic Input/Output

So far, the data sources used in our examples were simple, hard-coded `QueueSources`. Obviously, the events in real-world use cases are more likely to come from somewhere else: a file, the program's standard input, or some other source. BeepBeep's `io` package provides a few functionalities for connecting processor chains to the outside world.

READING FROM A FILE

Consider for example a text files containing single numbers, each on a separate line:

```
3
1
4
1
5
9
2.2
```

The `ReadLines` processor takes a Java `InputStream`, and returns as its output events each text line that can be extracted from that stream. Pulling from a `ReadLines` processor is then straightforward:

```
InputStream is = LineReaderExample.class.getResourceAsStream("pi.txt");
ReadLines reader = new ReadLines(is);
ApplyFunction cast = new ApplyFunction(Numbers.numberCast);
```

```
Connector.connect(reader, cast);
Pullable p = cast.getPullableOutput();
while (p.hasNext())
{
    Number n = (Number) p.next();
    System.out.println(n + "," + n.getClass().getSimpleName());
}
p.pull();
```

✎

A few important observations must be made from this code sample. The first is that since we are reading from a file, eventually the `ReadLines` processor will reach the end of the file, and no further output event will be produced when pulled. Therefore, the `Pullable` object must repeatedly be asked whether there is a new output event available. This can be done using the `hasNext` method. This method returns `true` when a new event can be pulled, and `false` when the corresponding processor has no more events to produce. Therefore, in our code sample, the program stays in the loop until `hasNext` returns `false`.

Note also that instead of using the `pull` method, we use the `next` method to get a new event. Methods `pull` and `next` are in fact *synonyms*: they do exactly the same thing. However, the pair of methods `hasNext`/`next` makes a `Pullable` look like a plain old Java `Iterator`. As a matter of fact, this is precisely the case: although we did not mention it earlier, a `Pullable` does implement Java's `Iterator` interface, meaning that it can be used in a program wherever an `Iterator` is expected. This makes BeepBeep objects very handy to use inside an existing program, without even being aware that they actually refer to processor chains.

One last comment: the output events of `ReadLines` are *strings*. This implies that, if we want to pipe them into arithmetical functions, they must be converted into `Number` objects beforehand; forgetting to do so is a common programming mistake. A special function of utility class `Numbers`, called `NumberCast`, is designed especially for that. This function takes as input any Java `Object`, and does its best to turn it into a `Number`. In particular, if the object is a `String`, it tries to parse that string into either an `int` or, if that fails, into a `float`. In the code example, the output of `reader` is piped into an `ApplyFunction` processor that invokes this function on each event; the function is referred to by the static member field `Numbers.numberCast`.

The expected output of the program is:

```
3,Integer
1,Integer
4,Integer
1,Integer
5,Integer
9,Integer
2.2,Float
Exception in thread "main" java.util.NoSuchElementException
    at ...
```

Note how the first lines of the file have been cast as an `Integer` number; the last number could not be parsed as an integer, therefore it has been cast as a `Float`.

The last printed lines show that an exception has been thrown by the program. This is caused by the very last instruction in the code, which makes one last `pull` on `p`. However, this happens right after `p.hasNext()` returns false, which takes the program out of the loop. As we have said earlier, attempting to pull an event from a `Pullable` that has no more event to produce causes such an exception to be thrown. Yet another programming mistake is to disregard the return value of `hasNext` (or not even calling it in the first place) and attempting to pull from a source that has "run dry".

The processor chain in this program can be represented as follows:

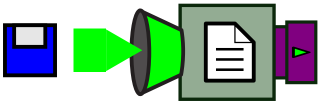


Figure 4.10: Reading lines from a text file with `ReadLines`.

This diagram introduces two new elements. First, the `ReadLines` processor is a box with a white sheet as its pictogram. As expected, the processor has one output pipe, which is painted in purple –the colour representing streams of `String` objects. Second, the processor seems to have an input pipe, but of a different shape than the ones seen earlier. This symbol does *not* represent a pipe, as can be confirmed by the fact that the input arity of `ReadLines` is zero. The funnel-shaped symbol rather represents a Java `InputStream` object. As we know, an `InputStream` can refer to an arbitrary source of bytes: a file, a network connection, and so on. Therefore, this symbol is intended to indicate that the

line reader takes its source of bytes from some outside source –more precisely, from something that is not a BeepBeep processor. BeepBeep's square pipes cannot be connected into funnels, and vice-versa. The light-green colour of the funnel indicates that the input stream provides raw bytes to the reader. The leftmost diskette symbol indicates that this particular input stream is connected to a file source.

READING FROM THE STANDARD INPUT

As seen earlier, we can read lines from a source of text by passing an `InputStream` to a `ReadLines` processor. However, it is possible to read from arbitrary streams of bytes, and in particular from the special system stream called the **standard input**. The standard input is an implicit stream present in every running program; external processes can connect to this stream and send bytes that can then be read by the program.

In Java, the standard input can be manipulated like any `InputStream`, using the static member field `System.in`. It could be sent to a `ReadLines` processor as we have done before; however, instead of complete lines of text ending with the newline character (`\n`), suppose we want to read arbitrary chunks of characters. This can be done by using another processor called `ReadStringStream`. The following program reads characters from the standard input and, using a `Print` processor, prints them back onto the standard output.

```
ReadStringStream reader = new ReadStringStream(System.in);
reader.setIsFile(false);
Pump pump = new Pump(100);
Thread pump_thread = new Thread(pump);
Connector.connect(reader, pump);
Print print = new Print();
Connector.connect(pump, print);
pump_thread.start();
while (true)
{
    Thread.sleep(10000);
}
```

✎

Since it works only in pull mode, and `Print` works only in `Push` mode, a `Pump` must be placed in between to repeatedly pull bytes from the input and push

them to the output. This can be represented graphically as follows:

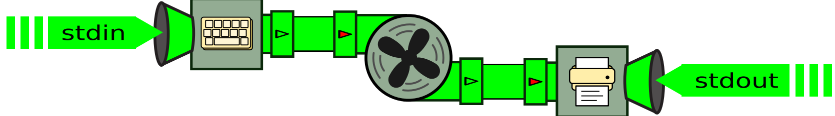


Figure 4.11: Reading characters from the standard input.

In this picture, the leftmost processor is the `StreamReader`. As you can see, it takes its input from the standard input; note how its left-hand side input has the "funnel" shape that represents system streams (and not BeepBeep pipes). A similar comment can be made for the `Print` processor, which was seen earlier. It receives input events, but as far as BeepBeep is concerned, does not produce any output events. Rather, it sends whatever it receives to the "outside world", this time through the `stdout` system stream. This is also what the `Print` processor does in examples from the previous chapters; however, the "stdout" output which was implicit in those examples is explicitly written here, in the diagram.

This program can be compiled as a runnable JAR file (e.g. `read-stdin.jar`) and tried out on the command line. Suppose you type:

```
$ java -jar read-stdin.jar
```

Nothing happens; however, by typing a few characters and pressing `Enter`, one should see the program reprint exactly what was typed (followed by a comma, as the `Print` processor is instructed to insert one between each event).

Let's try something slightly more interesting. At a Unix-like command prompt, one can create a *named pipe* using a command called `mkfifo`. Let us create one with the name `mypipe`:

```
$ mkfifo mypipe
```

Now, let us launch `read-stdin.jar`, by redirecting `mypipe` into its standard input:

```
$ cat mypipe > java -jar read-stdin.jar
```

By opening another command prompt, one can then push characters into `mypipe`; for example using the command `echo`. Hence, if you type

```
$ echo "foo" > mypipe
```

the string `foo` should immediately be printed at the other command prompt. This happens because `read-stdin.jar` continuously polls its standard input for new characters, and pushes them down the processor chain whenever it receives them.

As you can see, the use of stream readers in BeepBeep, combined with system pipes on the command line, makes it possible for BeepBeep to interact with other programs from the command line, in exactly the same way Unix programs can be connected into each other.

This can be used to read a file. Instead of redirecting a named pipe to the program, one can use the `cat` command with an actual filename:

```
$ cat somefile.txt > java -jar read-stdin.jar
```

This will have the effect of reading and pushing the entire contents of `somefile.txt` into the processor chain.

SEPARATING THE INPUT INTO TOKENS

In general, reading from an external source is done in "chunks" of bytes that do not necessarily correspond to the boundaries of data elements. For example, suppose that `somefile.txt` contains a list of words separated by commas:

```
the,quick,brown,fox,jumps,over,...
```

A `StreamReader` processor connected to this file will output events in the form of character strings; however, the commas that are present in the file have no special meaning for this processor. Therefore, the output of the `StreamReader` is likely to be made of pieces of text like this:

```
the,qui
ck,brown,fo
x,jumps,o
ver,...
```

As you can see, words can be cut across two successive chunks, and a single chunk may contain more than one word. Extra work must be done in order to reconstruct words out of these events: this involves gluing together some events, and cutting the strings to re-align them with the comma delimiters. In other words, we need to re-create "tokens" out of the sequence of strings, through a process that we call **tokenization**.

BeepBeep's `FindPattern` can be used for this purpose. The `FindPattern` processor receives pieces of text as its input, and produces for its output the portions of the text that match a specific *regular expression* as individual events. (A regular expression –"regex" for short– is a way of specifying a pattern of characters that is commonly used in programming.)

The following program shows a simple use of `FindPattern`:

```
ReadStringStream reader = new ReadStringStream(System.in);
reader.setIsFile(false);
Pump pump = new Pump(100);
Thread pump_thread = new Thread(pump);
Connector.connect(reader, pump);
FindPattern feeder = new FindPattern("(.*?),");
Connector.connect(pump, feeder);
Print print = new Print().setSeparator("\n");
Connector.connect(feeder, print);
pump_thread.start();
while (true)
{
    Thread.sleep(10000);
}
```

✎

Upstream, the `FindPattern` processor is connected to a `Pump`, itself connected to a `ReadStringStream` that polls the standard input. Downstream, `FindPattern` is connected to a `Print` processor that will display its output on the console. The pattern to look for, in this case, is represented by the regex "`(.*?),`". This expression matches any number of characters (`.*?`), followed by a comma. The parentheses do not match actual characters, but rather represent what is called a *capture group*; when a piece of text matches the whole regex, the `FindPattern` processor only outputs the part inside the capture group. Here, this indicates that the trailing comma will be taken out of each output event.

Let us compile this program as a runnable file called `read-tokens.jar`, and run it by redirecting the named pipe `mypipe` as in the previous example:

```
$ cat mypipe > java -jar read-tokens.jar
```

From a second command line window, you can now push strings to the program by `echo`ing them to `mypipe`:

```
$ echo "abc,def," > mypipe
```

The program will output

```
abc
def
```

Note here how each of "abc" and "def" have been printed on *two* separate lines. This is because the processor broke the input string into two events, since there are two commas indicating the presence of two tokens. This also means that `feeder` waits for the comma before outputting an event; hence writing the following command will result in no output.

```
$ echo "gh" > mypipe
```

The processor buffers the character string until it sees the desired pattern. Typing:

```
$ echo "i,jkl," > mypipe
```

will produce

```
ghi
jkl
```

READING FROM AN HTTP REQUEST

Instead of reading local files, it is also possible to obtain text from a remote source using the HTTP protocol. The `HttpGet` processor is a source which, when pulled, sends an HTTP GET request to a predefined URL, and returns as its output the contents of the response to the request. For example, the following program polls a URL every 10 seconds and prints the response to the console.

```
HttpGet get = new HttpGet("http://example.com/some-url");
Pump pump = new Pump(10000);
Thread pump_thread = new Thread(pump);
Connector.connect(get, pump);
Print print = new Print();
Connector.connect(pump, print);
pump_thread.start();
while (true)
{
```

```
    Thread.sleep(10000);
}
```

✎

The interest of this technique lies in the fact that the resource at the end of the URL does not need to be a static file. If the server that replies to the request returns content that changes over time, a repeated polling can be used as a dynamic source of events.

Soft vs. Hard Pulling

So far, `Pullable` objects were used as ordinary Java iterators. The `hasNext` method is used to ask whether a new event is available; if the answer is `true`, we can then use `pull` to fetch this new event, with the guarantee that there is indeed a new event to fetch. On the contrary, if the answer is `false`, this means that the processor to which the `Pullable` is attached has stopped producing events for good. As for an iterator over a normal collection of objects, it is useless to try to call `hasNext` at a later time: no new event will ever come out.

What happens when a processor *may* have more events to produce, but none is immediately available? In such a case, the correct answer to `hasNext` is neither `true` (there is no event available right now) nor `false` (the stream is not necessarily over). This is why BeepBeep `Pullable`s provide two ways for querying and pulling events: the “hard” and the “soft” methods. To illustrate the difference between the two, consider the following code:

```
QueueSource source = new QueueSource().loop(false);
source.setEvents(0, 1, 2, 3);
CountDecimate decim = new CountDecimate(2);
Connector.connect(source, decim);
Pullable p = decim.getPullableOutput();
```

✎

In this simple example, a source of eight numbers is connected to a `Count-Decimate` processor that will keep every third event. The source is configured *not* to loop to the first event of its list once the first ten have been output.

HARD PULLING

Hard pulling is the pull mode we have used so far. Let us call `hasNext` and `pull` a few times on this chain, as follows:

```
for (int i = 0; i < 5; i++)
{
  boolean b = p.hasNext();
  System.out.println(b);
  if (b == true)
  {
    System.out.println(p.pull());
  }
}
```

✎

The program prints, as expected:

```
true
0
true
2
false
false
false
```

It is worth taking some time to understand precisely what happens under the hood in this example. The sequence of method calls is summarized in Figure 4.12

In the first call to `hasNext` on `p`, the `Pullable` object asks `decim` whether it can produce a new output event. This triggers `decim` calling `hasNext` on `source`'s own pullable; `source` does have an event to output, so `decim` then calls `pull` to receive the number 0. Since 0 is `decim`'s first event, it can be output, so it places it in its output queue and the call to `hasNext` returns `true`, the first line of the program's output.

The next instruction is a call to `p`'s `pull` method. Since the number 0 is already waiting in `decim`'s output queue, `p` simply removes and returns it: this is the second line of the program's output.

The program then proceeds to a second turn of the loop. Method `hasNext` is again called on `p`; in turn, processor `decim` calls `hasNext` on `source`'s pullable

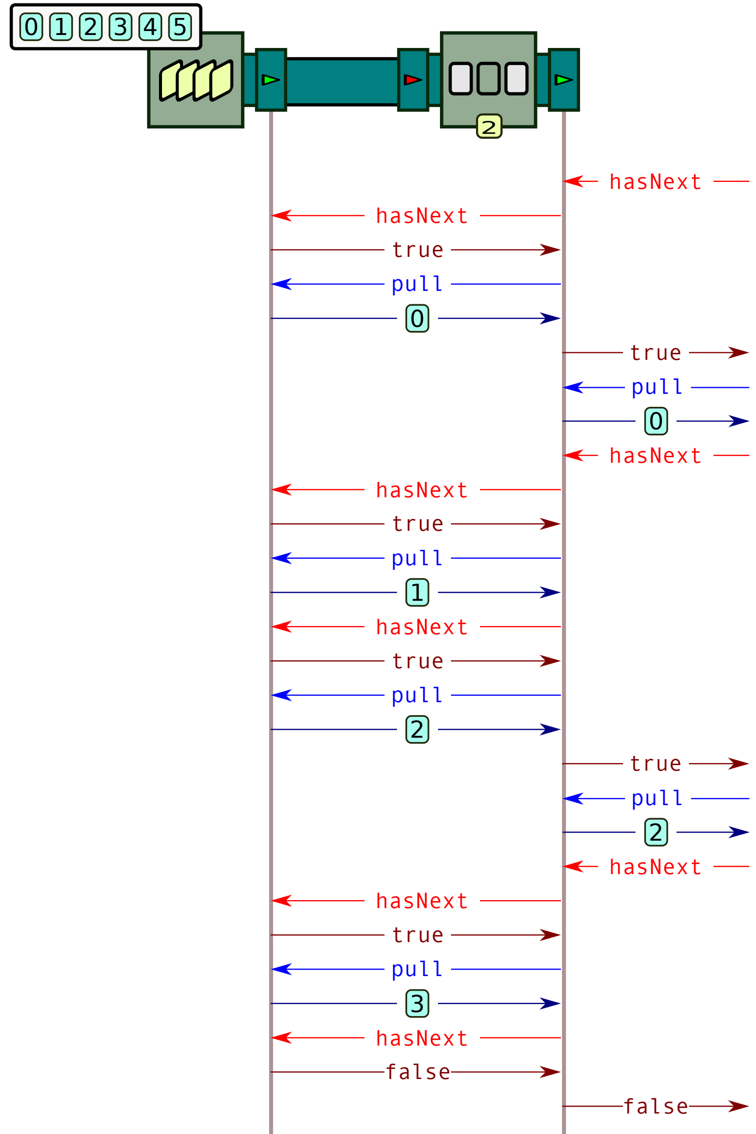


Figure 4.12: The sequence of method calls that occurs when hard pulling is used.

and receives the number 1. However, what happens after is different. Since `decim` only outputs every third event, it cannot output 1 and has to discard it. But then, this means that `decim` still does not know if it can output a new event. Therefore, it calls `hasNext` and then `next` on `source`'s pullable and receives the number 3; this time, the event can be output. It places the number in the output queue, and the call to `p`'s `hasNext` returns `true`. This corresponds to the third line of the output.

As noted early on in this book, a single call to `hasNext` on `p` has resulted on `decim` pulling three events from `source` before returning `true`. In hard pulling, a processor keeps pulling on its upstream processor until one of two things happen:

- It can produce an output event; in this case, the call to `hasNext` returns `true`.
- It is told by the upstream processor that no more events will ever come (i.e., its own call to `hasNext` on the upstream processor returns `false`); in this case, the call to `hasNext` returns `false`.

The rest of the program proceeds in the same way. Note that, after outputting the number 6, the call to `p`'s `hasNext` that follows returns `false`. Indeed, `decim` queries and obtains the number 7 from `source`, which it discards; the next call to `hasNext` on `source`'s pullable returns `false` (the source will never output a new event), which entails that `decim` will never output a new event. Object `p` remembers this, so that on any subsequent call to `hasNext`, it does not even bother to ask `decim` for new events and directly returns `false`.

SOFT PULLING

Soft pulling behaves a little differently. To illustrate this, we shall use the same processor chain as above, but replace calls to `hasNext` and `pull` by calls to two new methods: `hasNextSoft` and `pullSoft`.

```
for (int i = 0; i < 5; i++)
{
  Pullable.NextStatus s = p.hasNextSoft();
  System.out.println(s);
  if (s == Pullable.NextStatus.YES)
  {
    System.out.println(p.pullSoft());
```

```
  }
}
```

✎

We can observe that `hasNextSoft`, contrary to `hasNext`, does not return a Boolean, but rather a special value of type `NextStatus`. This type is actually an enumeration of three symbolic constants: `YES`, `NO` and `MAYBE`. A call to `hasNextSoft` that returns `YES` or `NO` has the same meaning as a call to `hasNext` returning `true` or `false`, respectively. When `YES` is the answer, a new event is available and ready to be pulled. When `NO` is the answer, no new event will ever come out of this `Pullable` object.

However, `hasNextSoft` can also return `MAYBE`, which indicates that the underlying processor does not have a new event to output, but *may* have one on a subsequent call to `hasNextSoft`. Let us look at the output of our modified program:

```
YES
0
MAYBE
YES
2
MAYBE
NO
```

The sequence of method calls that occurs is illustrated in the Figure 4.13

The beginning of the sequence unfolds in a very similar way to the "hard" example. The call to `hasNextSoft` on `p` triggers a call to `hasNextSoft` on `source`'s Pullable; since an event is available, it returns the value `YES`; `p` then calls `pullSoft`, and obtains the number 0. Processor `decim` produces an output event (i.e. 0), which is then relayed by `p` to the main program.

The sequence begins to differ at the second call to `hasNextSoft` on `p`. Again, `p` queries and receives the number 1 form `source`; as before, `decim` does not produce an output event from it. The difference lies in the fact that, rather than asking for another input event, `p` returns immediately with the value `MAYBE`. This is consistent with the definition of this value given earlier: `decim` cannot produce an output event right away, but the possibility that it emits an event on a subsequent call to `hasNextSoft` is not ruled out. Indeed, the next call to `hasNextSoft` has `decim` produce an output event, so its return value is `YES`; the next call to `pullSoft` returns the event 2.

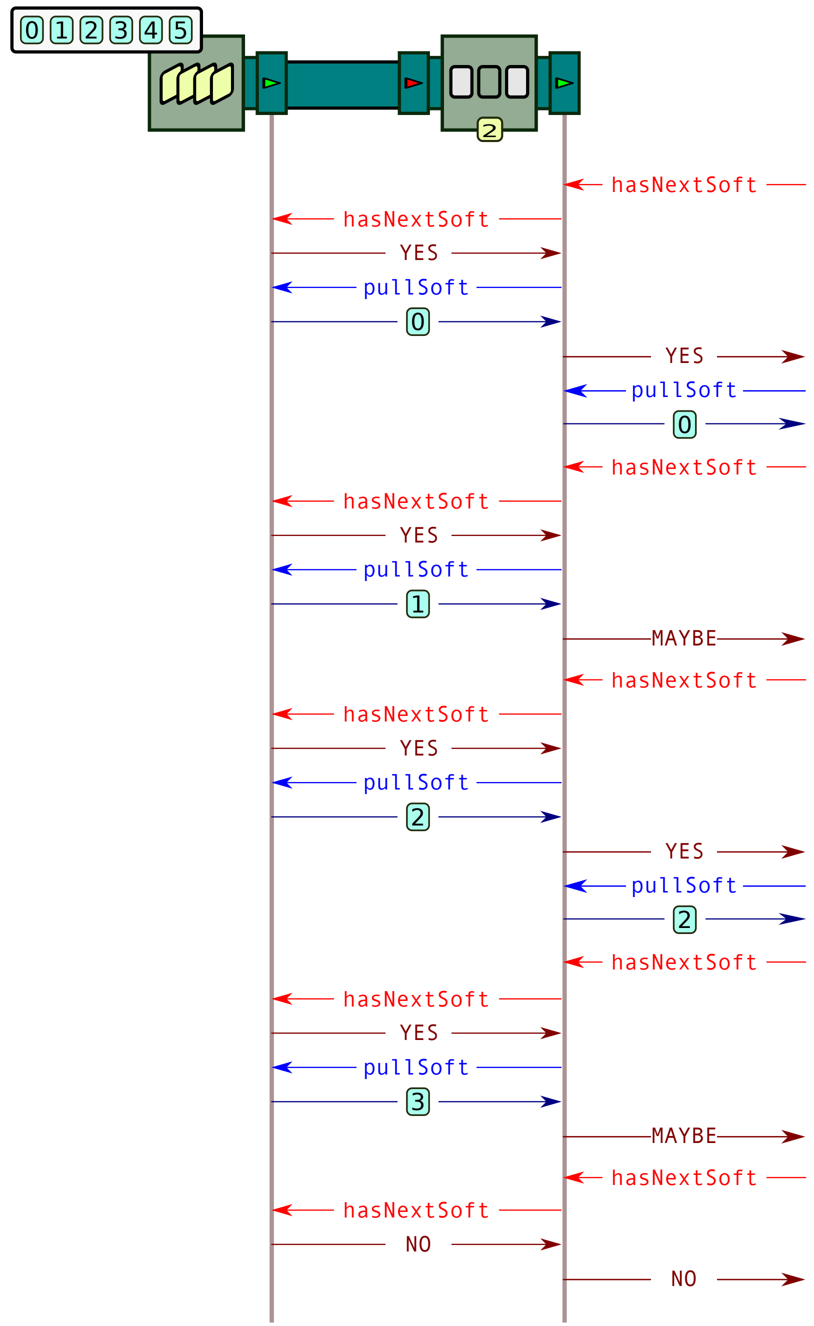


Figure 4.13: The sequence of method calls that occurs when soft pulling is used.

Calling `hasNextSoft` one more time on `p` produces again the value `MAYBE`, as `decim` cannot produce an output event from the number 3 it received from `source`. Finally, in the last call to `hasNextSoft`, `p` calls `hasNextSoft` on `source`, but this time receives the answer `NO`. This means that `source` will never be able to produce a new output event again, and so the call on `p` also returns `NO` for the same reason.

As you can see, using hard and soft pulling ultimately produces the same stream of output events in the main program. The difference lies in how these events are queried. Intuitively, soft pulling can be seen as doing exactly "one turn of the crank" on the whole chain of processors: every processor in the chain asks exactly once for an input event from upstream, which may or may not lead to the production of an output event. In contrast, hard pulling can be seen as doing soft pulling as long as needed to obtain an output event. Another way of seeing this is to say that soft pulling always returns, but maybe note with a new event; hard pulling, on its side, blocks until an output event is available.

For the same processor, mixing calls to soft and hard methods is discouraged. As a matter of fact, the `Pullable`'s behaviour in such a situation is left undefined.

Partial Evaluation

We said in Chapter 2 that BeepBeep's processing is *synchronous*: a processor takes no action until a complete event front is ready to be consumed. This assertion should be modified slightly, as there exists a single situation where this is not the case.

Consider a processor `m` that receives two streams of numbers, and multiplies their value. This processor has two input `Pushable` objects, `p1` and `p2`, which correspond to the pipes of its two input streams. Suppose we push value 3 into `p1`. Obviously, `m` does not have enough data to produce an output, as it is still waiting for an event from its second input pipe. This is consistent with the principle of synchronous processing we just discussed. Suppose, however, that we pushed the number `0` into `p1`. One may recall the special property of multiplication that `0` multiplied by any number equals `0`. In other words, there is no need to wait for a number on `p2`, as we already know that the output will be `0`. It would be useful if BeepBeep could accommodate these exceptional

situations, and allow functions to be *partially evaluated*.

There exists a variant of the `ApplyFunction` processor that works exactly in this way: it is called `ApplyFunctionPartial`. Contrary to the standard `ApplyFunction` processor, this variant can attempt to evaluate a function on incomplete input fronts, by filling the missing values with `null`. Consider the following code snippet:

```
ApplyFunctionPartial af =
    new ApplyFunctionPartial(Numbers.multiplication);
Print print = new Print();
Connector.connect(af, print);
Pushable p1 = af.getPushableInput(0);
Pushable p2 = af.getPushableInput(1);
```

✎

This program evaluates the `Multiplication` function through an `ApplyFunctionPartial` processor, and prints the result to the console. Pushing numbers other than 0 makes the processor behave like `ApplyFunction`; therefore, the next two lines of the program result in the number `3` being printed, as expected:

```
p1.push(3);
p2.push(1);
```

✎

The pair of input events (3,1) corresponds to the first input front. Then, pushing 0 on `p1` results in `af` immediately pushing the number 0, which is printed at the console:

```
p1.push(0);
```

✎

This call to `push` results in `af` receiving a first event in the second input front. Even if the matching event on `p2` is not yet available, the multiplication function can already produce the value 0. However, it is important to note that `ApplyFunctionLazy`, even if it immediately outputs an event, still keeps track of the relative position of events in each input pipe. Let us examine the remaining lines of the program:

```
p1.push(6);
p2.push(9);
```

```
p2.push(5);
```

✎

The first line pushes an event on `p1`, which corresponds to the third input front. Since this value is not 0, and that the corresponding event on `p2` is not yet available, the processor outputs nothing. The second line pushes the value 9 on `p2`. This corresponds to the second event front, which is now complete: (0,9). However, since an output value has already been produced for this input front, the event is simply ignored. Finally, the third line pushes the value 5 on `p2`; this corresponds to the third event front, which is also complete: value 5 on `p2` is matched with value 6 on `p1`, and their product (30) can be computed by the processor.

As one can see, given the input streams 3,0,6 on `p1` and 1,9,5 on `p2`, the output produced by `ApplyFunctionPartial` is indeed the pairwise product 3,0,30. In this respect, this processor produces the same result as `ApplyFunction`. However, it differs from it in the *moment* at which these events are output, which can, in some occasions, occur earlier.

In BeepBeep's core, a handful of functions support partial evaluation: multiplication, but also the Boolean connectives `And`, `Or` and `Implies`. Using other functions inside `ApplyFunctionPartial` has no special effect; in such cases, the processor behaves like `ApplyFunction`. Notice also that partial evaluation must be explicitly enabled by encasing a function inside `ApplyFunctionPartial`; this means that, even if a function supports partial evaluation, this feature will not be used inside a regular `ApplyFunction` processor. This is done by design, since partial evaluation has a higher computational cost than regular evaluation. With a function of input arity k, `ApplyFunctionPartial` attempts to partially evaluate a function every time an event arrives on an input front; this may turn out to be more costly than waiting for the k events to be available before evaluating the function once.

The State of a Processor

We mentioned a few times that the main distinction between functions and processors is the fact that the latter are *stateful*. That is, given the same inputs, a function always returns the same output; in contrast, the output produced by a processor for an event may depend on what other events have been seen

before. As a consequence, a `Processor` object must have some memory of the past –hence the term “stateful”.

Consider for example a `CountDecimate` processor instructed to keep one event every *k*. In order to correctly perform its task, this processor must keep a record of the number of events that have been discarded since the last time an output event was produced. This value is incremented by one, modulo *k*, every time a new input event is received; when the value reaches 0, a new output event is produced. Hence, the *state* of a `CountDecimate` processor corresponds to the current value of its counter. Typically, given the same input event *and the same state*, a processor is expected to always behave in the same way, i.e. produce the same output, if any. Depending on the processor type, the state may be a single numerical value, or something more complex. For example, in a `Window` processor, the current state consists of the input events that have been accumulated in partial windows.

Each `Processor` subclass is expected to have a well-defined initial state, in which all instances are expected to start (unless the class' constructor accepts arguments that may modulate that initial state). In addition, processors can be reset to their initial state using a method called `reset()`.

Let us take as an example a `Window` processor that computes the cumulative sum of a sliding window of three events, as in the following program:

```
Window w = new Window(new Cumulate(
    new CumulativeFunction<Number>(Numbers.addition)), 3);
Connector.connect(w, new Print());
Pushable p = w.getPushableInput();
p.push(3).push(1).push(4).push(1).push(6);
w.reset();
p.push(2).push(7).push(1).push(8).push(3);
```

✎

As expected, the first two calls to `push` do not send anything to the `Print` processor, since three input events are required for `Window` to output something. The remaining three calls to `push` produce the first three numbers printed on the console:

```
8,6,11,
```

Then, the window processor `w` is reset to its initial state, and new events are pushed to it. Notice how, again, the first two calls to `push` after the reset has

been made are not printing anything. Processor `w` has been put back into its initial state, meaning in this case that its window is cleared of all its events, and its associated `Cumulate` processor is also reset to a count of zero. This is why the next three calls to `push` print on the console:

```
10,16,12,
```

In the case of a `GroupProcessor`, a call to `reset` has for effect of calling `reset` on all the processors it contains, as well as emptying all the input and output queues of these processors.

In addition to objects and member fields specific to the implementation of each `Processor` descendent, BeepBeep processors also have some more memory elements that are carried by every processor instance. We discuss them in the following.

NUMERICAL IDENTIFIER

Each `Processor` object has a unique numerical identifier, called the *processor ID*. In the current implementation of BeepBeep, identifiers start at zero when a program is launched, and are incremented by one every time the constructor of class `Processor` is called (in other words, every time a new processor of any kind is created). The assignment of an ID to each processor is synchronized, meaning that race conditions are avoided in the case where processors are instantiated from multiple threads in parallel.

The ID of a processor has no special meaning, and you will seldom have to use this value when writing processor chains in your daily work. Processor IDs are used by the BeepBeep library itself, mostly for two purposes:

- correctly copying `GroupProcessors` when the `duplicate` method is called (see below);
- tracking the relationship between input and output events throughout a chain of processors (a very embryonic feature called *provenance*, which will be further developed in future versions of the software).

Nevertheless, IDs can be queried using a public method called `getId()`. The following code shows an example of this:

```
QueueSource source = new QueueSource().loop(false);
source.setEvents(0, 1, 2, 3);
CountDecimate decim = new CountDecimate(2);
```

```
System.out.println("ID of source: " + source.getId());
System.out.println("ID of decim: " + decim.getId());
```

✎

The program simply create two processors and prints their respective ID:

```
ID of source: 0
ID of decim: 1
```

We insist on the fact that *every* instance has a distinct ID. Consider the following class, which creates a `GroupProcessor` containing a single `Passthrough`:

```
public static class MyGroup extends GroupProcessor
{
  protected Passthrough pt;
  public MyGroup()
  {
    super(1, 1);
    pt = new Passthrough();
    addProcessor(pt);
    associateInput(0, pt, 0);
    associateOutput(0, pt, 0);
  }
  public Passthrough getInstance()
  {
    return pt;
  }
}
```

✎

Let us create an instance of this group, and look at the IDs of the processors:

```
MyGroup g1 = new MyGroup();
System.out.println("ID of g1: " + g1.getId());
System.out.println("ID inside g1: " + g1.getInstance().getId());
```

✎

The output of this program is:

```
ID of g1: 0
ID inside g1: 1
```

As you can see, the `GroupProcessor` in itself has its own ID (0), and the instance of `Passthrough` it contains has a distinct ID. Let us continue the pro-

gram and create a second instance of `MyGroup`:

```
MyGroup g2 = new MyGroup();
System.out.println("ID of g2: " + g2.getId());
System.out.println("ID inside g2: " + g2.getInstance().getId());
```

✎

The next two printed lines are:

```
ID of g1: 2
ID inside g1: 3
```

As you can see, the uniqueness of processor IDs is *global* across the entire run of a program. Even for multiple copies of the same `GroupProcessor`, every processor it contains is given an ID different from that of any other processor. However, you *cannot* assume that the same processor instance gets the same ID every time the program is run: this may depend on the interlacing of threads, or any other source of non-determinism (such as enumerations over unordered collections, user input, etc.).

CONTEXT

In addition, each processor instance is also associated with a **context**. A context is a persistent and modifiable map associating names to arbitrary objects. A processor's context can be manually modified using method `setContext`, as in the following example:

```
ApplyFunction af = new ApplyFunction(new FunctionTree(
        Numbers.addition,
        StreamVariable.X,
        new ContextVariable("foo")));
Connector.connect(af, new Print());
af.setContext("foo", 10);
Pushable p = af.getPushableInput();
p.push(3);
af.setContext("foo", 6);
p.push(4);
```

✎

An `ApplyFunction` processor is created, and an association between the key "foo" and the number 10 is added to the processor's context object. This context

can be referred to in a `FunctionTree` by using a `ContextVariable`. Here, such a variable is created and is instructed to fetch the value associated to key "foo" in the current processor's context. Therefore, the output of the program is:

```
10,13,
```

Note how the context can be modified by further calls to `setContext`. If a processor requires the evaluation of a function, the current context of the processor is passed to the function. Hence, the function's arguments may contain references to names of context elements, which are replaced with their concrete values before evaluation. Basic processors, such as those described so far, do not use context. However, some special processors defined in extensions to BeepBeep's core (the Moore machine and the first-order quantifiers, among others) manipulate their `Context` object.

Duplicating Processors

Occasionally, it may be useful to create a copy of an existing processor instance. This process is called **duplication**, and it is done using a processor's method `duplicate()`. There are two types of duplication: *stateless* and *stateful*.

STATELESS DUPLICATION

The following example shows how to perform what is called *stateless* duplication:

```
Cumulate sum1 = new Cumulate(
    new CumulativeFunction<Number>(Numbers.addition));
Connector.connect(sum1, new Print()
    .setPrefix("sum1: ").setSeparator("\n"));
Pushable p_sum1 = sum1.getPushableInput();
p_sum1.push(3).push(1).push(4);
```

✎

In this example, a `Cumulate` processor is connected to a `Print` sink, and a few events are pushed to it. The first part of the program prints, as expected:

```
sum1: 3
sum1: 4
sum1: 8
```

Let us now use method `duplicate()` to create a new copy of `sum1`, and push events to it as well:

```
Cumulate sum2 = (Cumulate) sum1.duplicate();
Connector.connect(sum2, new Print()
    .setPrefix("sum2: ").setSeparator("\n"));
Pushable p_sum2 = sum2.getPushableInput();
p_sum2.push(2).push(7).push(1);
```

✎

The next three lines the program prints are:

```
sum2: 2
sum2: 9
sum2: 10
```

This output reveals two things. First, `sum2` is a new `Cumulate` processor that adds numbers; this is indeed a copy of `sum1`. Second, `sum2` accumulates numbers into a sum of its own: it does not keep on adding to the numbers that were accumulated by `sum1` at the moment it was duplicated. Indeed, the default behaviour of `duplicate` is to create a new processor copy, and to place it into its *initial state* –that is, the state in which the processor would be if we called its constructor directly. This is why it is called *stateless* duplication: the current state of the original processor is not preserved.

However, when a processor is duplicated, its *context* is duplicated as well. To illustrate this, let us create a processor that applies a simple function:

```
ApplyFunction f1 = new ApplyFunction(new FunctionTree(
    Numbers.addition, StreamVariable.X, new ContextVariable("foo")));
Connector.connect(f1, new Print()
    .setPrefix("f1: ").setSeparator("\n"));
Pushable p_f1 = f1.getPushableInput();
f1.setContext("foo", 10);
p_f1.push(3).push(1);
ApplyFunction f2 = (ApplyFunction) f1.duplicate();
Connector.connect(f2, new Print()
    .setPrefix("f2: ").setSeparator("\n"));
Pushable p_f2 = f2.getPushableInput();
p_f2.push(2).push(7).push(1);
```

✎

The function adds the value of a processor's context variable called "foo" to

each input event. In the beginning, this variable is set to the value 10 by a call to `setContext` on `f1`. Processor `f1` is then duplicated, and a few more events are pushed on its copy `f2`. The output of the program is:

```
f1: 13
f1: 11
f2: 12
f2: 17
f2: 11
```

This shows that the value of "foo" (10) has been transferred over from `f1` to its duplicate `f2`. From then on, `f1` and `f2` have separate context objects; changing the value of "foo" in `f1`'s context has no effect on `f2`, and vice versa. Keep in mind that duplication is like any other processor instantiation, and that the duplicated processor always has a distinct numerical ID, regardless of everything else.

STATEFUL DUPLICATION

It is also possible to duplicate a processor *and* its state at the same time. To this end, method `duplicate` accepts an optional Boolean argument; if set to `true`, this will instruct to create a copy of the process, and to place that processor in the same state as the original (instead of its initial state). Let us examine the difference by revisiting our original example on duplication, and adding the parameter `true` to the call to `duplicate`:

```
Cumulate sum2 = (Cumulate) sum1.duplicate(true);
```

✎

The output of the program becomes the following; notice how `sum2`'s count does not start at 0, but rather at `sum1`'s count at the moment it was duplicated.

```
sum1: 3
sum1: 4
sum1: 8
sum2: 10
sum2: 17
sum2: 18
```

Duplication has uses for single processor instances, but proves even more handy when manipulating a `GroupProcessor`. In this case, the `duplicate` method takes care of creating a copy of the group; moreover, it duplicates every processor contained in that group, and re-pipes these copies to match exactly the way they are piped in the original. This is in line with the intent that a `GroupProcessor` makes a set of processors connected together behave as if they were a single box. When users call `duplicate` on an instance of a `Processor` object, they do not have to care whether they are duplicating a single processor, or a complex chain of processors encapsulated into a group.

To make things more concrete, let us examine this code example:

```
QueueSource source1 = new QueueSource().setEvents(3, 1, 4, 1, 5);
GroupProcessor gp1 = new GroupProcessor(1, 1);
{
  Stutter st = new Stutter(2);
  Cumulate sum = new Cumulate(
      new CumulativeFunction<Number>(Numbers.addition));
  Connector.connect(st, sum);
  gp1.addProcessors(st, sum);
  gp1.associateInput(0, st, 0);
  gp1.associateOutput(0, sum, 0);
}
Connector.connect(source1, gp1);
Pullable p_gp1 = gp1.getPullableOutput();
System.out.println(p_gp1.pull());
```

✎

We first create a `GroupProcessor` that encapsulates a `Stutter` processor connected to a `Cumulate`. The `Stutter` processor simply repeats each input event a specified number of times (here, 2). We then connect this group to an upstream source of numbers, and proceed to pull one event out of this chain. Without surprise, the output that is printed at the console is `3`.

The next part of the program will duplicate `gp1` into `gp2`, connect it to a new upstream source of numbers, and pull one event from `gp2`:

```
QueueSource source2 = new QueueSource().setEvents(2, 7, 1, 8);
GroupProcessor gp2 = gp1.duplicate(true);
Connector.connect(source2, gp2);
```

```
Pullable p_gp2 = gp2.getPullableOutput();
System.out.println(p_gp2.pull());
```

✎

The second line that is printed at the console is `6`. To understand how we got this number, we must have a look at what occurred under the hood. The first call to `pull` on `gp1` triggered a call to `pull` on the group's `sum` processor, which in turn triggered a call to `pull` on `stutter`, and finally a call to `pull` on `source1`. In return, the processor `stutter` received the event `3`, and placed *two* events in its output queue (the number `3` twice). Processor `sum` took the first of these two events, processed it and returned the value `3`, which in turn was returned by `gp1`.

Therefore, after the first call on `pull`, processor `stutter` inside `gp1` has one event waiting in its output queue (the second copy of the number `3`). The program then creates a duplicate of `gp1`, and the use of parameter `true` instructs BeepBeep to create a *stateful* duplicate. Therefore, `gp2` not only has processors connected in the same way as in `gp1`, but each processor inside has input and output queues that replicate the contents of the original. This means that the copy of `stutter` inside `gp2` also has the number `3` waiting in its output queue.

Once we understand this, the rest of the program is not surprising. Upon the call to `pull` on `gp2`, `sum` will fetch the second copy of number `3` from `stutter`; it will then add this value to its current total, which is 3 (don't forget that `sum` is also a stateful copy of its corresponding processor in `gp1`). This explains the return value of 6.

Copying the contents of input and output queues inside `GroupProcessors` is essential for such processors to behave like black boxes. In our example, we could create a single `Processor` object `p` that computes the cumulative sum of the double of each input event; this object would behave exactly like `gp1`. However, in our example program, a duplicate of `gp1` that would not preserve queue contents would return a different value than `p` on the second event (3 instead of 6) –meaning that this duplicate is not in the same internal state.

The discussion in this section is a bit technical, and BeepBeep is designed precisely so that you don't need to bother about these intricate details. However, it may sometimes be useful to get a deeper understanding of what is happening at a lower level of abstraction.

Exercises

1. Create a chain of processors that receives a stream of collections of integers, and outputs `true` for a collection if and only if it contains a number that corresponds to its size. For example, the set {1,3,6} is of size 3, and it contains the number 3, so the answer would be `true`. Do it…

a. Using a `FunctionTree`.
b. Without using a `FunctionTree`.

2. Create a chain of processors that receives three streams of numbers as its input. Its output should be a stream of *sets* of numbers. The output set at position i should contain the i-th element of each input stream, only if this element is positive. That is, if the first event of each stream is respectively -1, 3, 4, the first output set should be {3,4}.

3. Create a chain of processors that receives a stream of collections of numbers, and returns…

a. The average of each collection.
b. The largest number of each collection.

4. The `Strings` utility class in BeepBeep defines a `Function` object called `SplitString`. Use it to create a processor chain that receives a stream of arbitrary strings, and returns a stream made of each individual word, except those that start with the letter "a". For example, on the input event "this is an abridged text", the chain would produce the output events "this", "is", "text". (Hint: a simple solution involves the use of `Unpack`.)

5. Consider a stream of letters of the alphabet. Create a processor chain that always returns the number of occurrences of the letter that has been seen most often so far. For example, on the input stream a,b,a,c,c,b,a, the processor would return 1,1,2,2,2,2,3. (Hint: a possible solution involves `Slice`, `Cumulate`, `Maps.Values` and `Numbers.max`, among others.)

6. Create a processor chain that reads an HTML file as input, and counts how many times each HTML tag appears in the document. (Hint: use a `FindPattern` and a `Slice`, among others.)

7. In the section about processor states, one of the examples applies the

`duplicate()` method on a `Cumulate` processor. Try the same process on other elementary processors, such as `Trim` or `CountDecimate`. Can you guess what their initial state is?

5 The Standard Palettes

A large part of BeepBeep's functionalities is dispersed across a number of *palettes*. These palettes are additional libraries (i.e. JAR files) that define new processors or functions to be used with BeepBeep's core elements. Each palette is optional, and has to be included in your project only if you need its contents.

This modular organization has three advantages. First, palettes are a flexible and generic way to extend the engine to various application domains, in ways unforeseen by its original designers. Second, they compose the engine's core (and each palette individually) relatively small and self-contained, easing the development and debugging process. Palettes have many purposes: reading special file types, producing plots, accessing a network, and so on. In the same way that the C programming language generally ships with a "standard" library, in this chapter, we explore a few "standard" palettes of BeepBeep that are more frequently used than others.

Tuples

Input files are seldom made of a single value per line of text. A more frequent file format is called **comma-separated values** (CSV). In such a format, each line contains the value of multiple **attributes**, separated by a comma. The following is an example of such a file:

```
# This is a simple file in CSV format

A,B,C
3,2,1
1,7,1
```

```
4,1,2
1,8,3
6,3,5
```

Blank lines and lines beginning with the hash symbol (#) are typically ignored (although the latter is not standard). The first non-ignored line in the file provides the *name* of each attribute. In the example above, the file defines three attributes named “A”, “B” and “C”. All the remaining lines of the file defines what are called **tuples**; a tuple is a data object that associates each attribute to a value. For example, the fourth line of the file defines a tuple that associates attribute A to value 3, attribute B to value 2, and attribute C to value 1. In other words, a CSV file is similar to a **table** in a relational database.

READING TUPLES

The following program reads a CSV file called `file1.csv`, and extracts tuples from this file one by one:

```
InputStream is = CsvReaderExample.class.getResourceAsStream("file1.csv");
ReadLines reader = new ReadLines(is);
TupleFeeder tuples = new TupleFeeder();
Connector.connect(reader, tuples);
Pullable p = tuples.getPullableOutput();
Tuple tup = null;
while (p.hasNext())
{
    tup = (Tuple) p.next();
    System.out.println(tup);
}
```

✎

The first two lines are now familiar: they consist of opening an `InputStream` on a file, and passing this stream to a `ReadLines` processor to read it line by line. The next instruction creates a new processor called a `TupleFeeder`. This processor receives lines of text, and returns on its output pipe `Tuple` objects. The rest of the program simply pulls and prints these tuples. The output of this program is:

```
((A,3),(B,2),(C,1))
((A,1),(B,7),(C,1))
((A,4),(B,1),(C,2))
```

```
((A,1),(B,8),(C,3))
((A,6),(B,3),(C,5))
```

As you can see from the format of the output, a tuple can also be seen as a set of attribute-value pairs. `Tuple` objects implement Java's `Map` interface; therefore, their contents can be queried just like any other associative map:

```
Object o = tup.get("A");
System.out.println(o + "," + o.getClass().getSimpleName());
```

✎

If `tup` refers to the last `Tuple` pulled from `tuples`, the previous lines of code will print:

```
6,String
```

Note that **the values in tuples produced by `TupleFeeder` are always strings**. The `TupleFeeder` does not attempt to cast a string into a number.

Graphically, this program can be represented as follows:

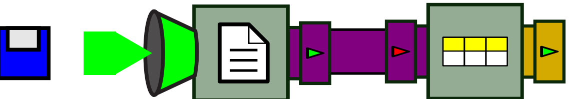


Figure 5.1: Converting strings into tuples.

This diagram introduces the symbol for the `TupleFeeder`, a pictogram on the box representing a tuple. It also shows the colour used to represent tuple feeds (brown/orange).

QUERYING TUPLES

The previous example has shown us how to read tuples, but not how to manipulate them. The `tuples` palette defines a few handy `Function` objects allowing us, among other things, to fetch the value of an attribute and also to merge tuples. From the same input file as above, let us create an output stream made of the sum of attributes A and B in each line. The following piece of code performs exactly that:

```
InputStream is = SumAttributes.class.getResourceAsStream("file1.csv");
ReadLines reader = new ReadLines(is);
```

```
TupleFeeder tuples = new TupleFeeder();
Connector.connect(reader, tuples);
Fork fork = new Fork(2);
Connector.connect(tuples, fork);
ApplyFunction get_a = new ApplyFunction(new FetchAttribute("A"));
Connector.connect(fork, 0, get_a, 0);
ApplyFunction get_b = new ApplyFunction(new FetchAttribute("B"));
Connector.connect(fork, 1, get_b, 0);
ApplyFunction cast_a = new ApplyFunction(Numbers.numberCast);
Connector.connect(get_a, cast_a);
ApplyFunction cast_b = new ApplyFunction(Numbers.numberCast);
Connector.connect(get_b, cast_b);
ApplyFunction sum = new ApplyFunction(Numbers.addition);
Connector.connect(cast_a, 0, sum, 0);
Connector.connect(cast_b, 0, sum, 1);
Pullable p = sum.getPullableOutput();
while (p.hasNext())
{
    System.out.println(p.next());
}
```

✎

This program is probably better explained through its graphical representation, as the following:

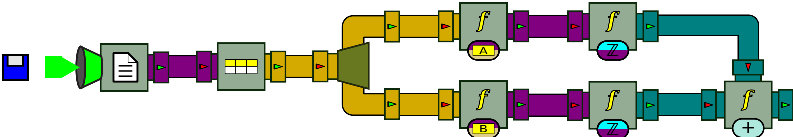


Figure 5.2: Adding two attributes in each tuple.

From a `ReadLines` processor, a `TupleFeeder` is instantiated. The stream of tuples is then forked along two branches. In the first branch, the value of attribute "A" for each tuple is extracted. This is done by using an `ApplyFunction` processor, and giving an instance of a new function called `FetchAttribute` to this processor. When instantiated, function `FetchAttribute` is given the name of the attribute to fetch in the tuple. This value (a String) is converted into a number and sent into an `ApplyFunction` processor that computes a sum. The same thing is done along the bottom branch for attribute "B". From the same input file as above, the output of this program is:

```
5.0
8.0
5.0
9.0
9.0
```

Indeed, it corresponds to the sum of A and B in each line. However, this processor chain is needlessly verbose. The successive application of all three functions can be collapsed into a single function tree, yielding this much simpler graph:

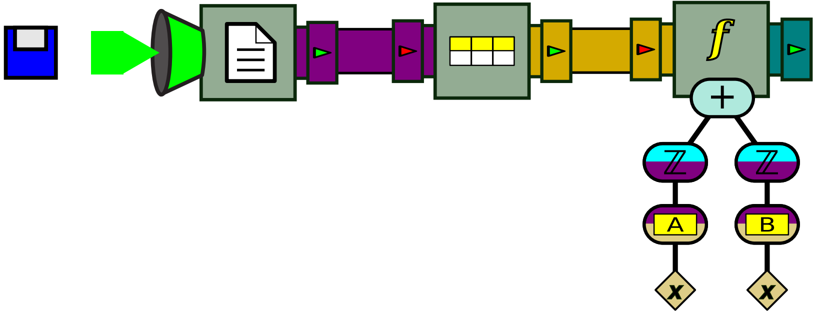


Figure 5.3: Adding two attributes in each tuple (alternate version).

As an exercise, we leave to the reader the task of writing this chain of processors in code.

OTHER TUPLE FUNCTIONS

The `tuples` palette provides a few other functions to manipulate tuples. Here, let us briefly describe a few of them:

- The function `ScalarIntoTuple` takes a scalar value x (for example, a number) and creates a tuple with a single attribute-value pair A=x. Here "A" is a name passed to the function when it is instantiated.
- The function `MergeTuples` merges the key-value pairs of multiple tuples into a single tuple. If two tuples have the same key, the value in the resulting tuple is that of one of these tuples, selected arbitrarily. However, if the tuples have the same value for their common keys, the resulting tuple is equivalent to that of a relational JOIN operation.

- The function `BlowTuples` breaks a single tuple into multiple tuples, one for each key-value pair of the original tuple. The output of this function is a *set* of tuples, and not a single one.
- The function `ExpandAsColumns` transforms a tuple by replacing two key-value pairs by a single new key-value pair. The new pair is created by taking the value of a column as the key, and the value of another column as the value. For example, with the tuple: {(foo,1), (bar,2), (baz,3)}, using "foo" as the key column and "baz" as the value column, the resulting tuple would be: {(1,3), (bar,2)}. The value of foo is the new key, and the value of baz is the new value. The other key-value pairs are left unchanged.

RELATIONAL DATABASES

We have already seen how a log of events stored in a file can be fed, line by line, to a BeepBeep processor chain and act as a pre-recorded event source. A BeepBeep palette allows users to do the same thing, this time using a relational database as the source of events. To this end, BeepBeep leverages Java's facilities for interacting with databases, regrouped under the name *Java Database Connectivity* (JDBC).

Suppose you have a local database server running on your machine. This server hosts a database called `mydb`, which itself contains a table called `mytable`. The contents of `mytable` are shown below:

Name	**Salary**
Fred Flintstone	1000
Barney Rubble	1200
Wilma Filntstone	1300
George Jetson	1100

It is possible to use the lines of this table as a source of events, each of which will consist of a `Tuple` object with the data of the corrresponding line. To this end, one uses a special BeepBeep processor called `JdbcSource`, which converts an SQL query sent to a database server into a stream of tuples. Consider the following program:

```
Connection conn = DriverManager.getConnection(
```

```
    "jdbc:mysql//localhost/mydb", "betty", "foo");
String query = "SELECT * FROM mytable";
JdbcSource src = new JdbcSource(conn, query);
Pullable p = src.getPullableOutput();
while (p.hasNext())
{
  Tuple t = (Tuple) p.pull();
  System.out.println(t);
}
```

✎

The first line of this program is standard JDBC: it creates a `Connection` object to a database, based on a JDBC URL, a user name and a password. In the present case, the program connects to a server hosted on the local machine (`localhost`), using the MySQL driver, on a database called `mydb`, using "betty" and "foo" as the user/password credentials. The next line defines a query to be executed on this database; in this case, this amounts to a simple `SELECT` statement picking all the columns and all the lines from table `mytable`.

The next instruction creates a BeepBeep `JdbcSource` object; this objects acts as a gateway between JDBC objects and BeepBeep processors. This source is given the database `Connection` object and the query to execute. From then on, `src` can be used like any other BeepBeep `Source` object. The next line obtains a reference to `src`'s `Pullable`, and repeatedly pulls events from it. As one can see by looking at the console, each event is indeed a `Tuple` object corresponding to a line of the result:

```
{"Name": "Fred Flintstone", "Salary" : "1000"}
{"Name": "Barney Rubble", "Salary" : "1200"}
...
```

It is useful to know that, under the hood, the `JdbcSource` does not call the database multiple times. It does so a single time, upon the first call to `pull`; this triggers the evaluation of the SQL query and the retrieval of its result as a JDBC `ResultSet` object. Each subsequent call to `pull` simply amounts to pulling one new line from the result set, until all lines have been enumerated.

Obviously, the basic `SELECT` statement we used in this example can be replaced by a more complex expression. Moreover, since tables are, by definition, unordered collections of tuples, the ordering in which `src` enumerates the tuples may vary from one execution to the next, unless an `ORDER BY` clause is present in the statement.

Finite-state Machines

Sometimes, a stream is made of events representing a sequence of "actions". It may be interesting to check whether these actions follow a predefined pattern, which stipulates in what order the actions in a stream can be observed to be considered valid. A convenient way of specifying these patterns is through a device called a *finite-state machine* (FSM). BeepBeep's FSM palette allows users to create such machines.

DEFINING A MOORE MACHINE

As a simple example, suppose that a log contains a list of calls on a single Java `Iterator` object. Typical method calls on an iterator are `next`, `hasNext`, `reset`, etc. Such a log could look like this:

```
hasNext
next
hasNext
hasNext
next
reset
...
```

The proper use of an iterator stipulates that one should never call method `next()` before first calling method `hasNext()`. The correct ordering of these calls can be expressed by a finite-state machine with three states, as in the following picture.

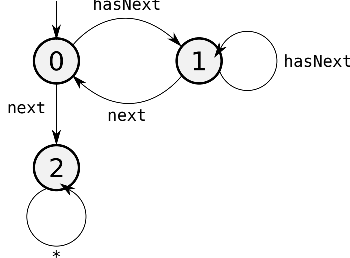


Figure 5.4: A finite-state machine representing the constraint that `next()` cannot be called before calling `hasNext()` first.

In this FSM, states are numbered 0, 1 and 2; transitions between states are

labelled with the method name they represent; for example, when the machine is in State 1, receiving a `next` event will make it move to State 0. One of these states is called the *initial state*, and is identified by an arrow that is unattached to any source state. In the present case, the initial state is 0. The "star" label on State 2's arrow indicates that this transition matches any incoming event.

In BeepBeep's FSM palette, finite-state machines are materialized by an object called `MooreMachine`; the origin of that name will be explained subsequently. The creation of the machine is made by the following code example:

```
MooreMachine machine = new MooreMachine(1, 1);
final int UNSAFE = 0, SAFE = 1, ERROR = 2;
machine.addTransition(UNSAFE, new FunctionTransition(
    new FunctionTree(Equals.instance,
        StreamVariable.X, new Constant("hasNext")), SAFE));
machine.addTransition(UNSAFE, new FunctionTransition(
    new FunctionTree(Equals.instance,
        StreamVariable.X, new Constant("next")), ERROR));
machine.addTransition(SAFE, new FunctionTransition(
    new FunctionTree(Equals.instance,
        StreamVariable.X, new Constant("next")), UNSAFE));
machine.addTransition(SAFE, new FunctionTransition(
    new FunctionTree(Equals.instance,
        StreamVariable.X, new Constant("hasNext")), SAFE));
machine.addTransition(ERROR, new FunctionTransition(
    new Constant(true), ERROR));
```

✎

The first step is to create an empty `MooreMachine`; this machine receives one stream of events as its input, and produces one stream of events as its output –hence the two `1` in the object's constructor. In a `MooreMachine`, each state must be given a unique numerical identifier. Rather than hard-coding these numbers, we adopt a cleaner approach, and define symbolic constants for the three states of the Moore machine. It is recommended that the actual numbers for each state form a contiguous interval of integers starting at 0. Here, we associate numbers 0, 1 and 2 to the constants `UNSAFE`, `SAFE` and `ERROR`, respectively.

We are now ready to define the transitions (i.e. the "arrows" between states) for this machine. This is just a tedious enumeration of all the arrows that are present in the graphical representation of the FSM. Adding a transition to the machine is done through a method called `addTransition()`. This

method must provide the number of the “source” state n_s, and a `Transition` object. There are multiple types of such objects, but a frequent subclass is the `FunctionTransition`. This object specifies:

- A `Function` f that determines when the transition should fire. This function must have the same input arity as the machine itself, and return a Boolean value.
- The number of the “destination” state n_d.

Intuitively, a `FunctionTransition` transition stipulates that when the machine is currently in state n_s and receives an event e, if $f(e)$ returns `true`, the machine shall move to state n_d. For example, the first line states that in State 0 (`UNSAFE`), if the incoming event is “hasNext”, go to State 1 (`SAFE`). The condition itself is expressed by creating a `FunctionTree` that checks if the incoming event (which is put into the `StreamVariable`) is equal to the `Constant` “hasNext”. By default, the first state number that is ever given to the `MooreMachine` object is taken as the initial state of that machine. So here, `UNSAFE` will be the initial state. In BeepBeep’s implementation of FSMs, there can only be one initial state.

The remaining instructions simply add the other transitions to the machine. A special remark must be made about State 2, which is a *sink state*; in other words, once you reach this state, you remain there forever. These states are typically used to indicate that the system has entered into an irrecoverable error condition. A possible way to say so is to define the condition on its only transition as the `Constant` true; it will fire whatever the incoming event may be.

These seven lines of code completely define our FSM. However, as it is, the machine is not instructed to output any event at any time. We mentioned earlier that this FSM is of a particular kind, called a *Moore machine*. Such a machine outputs a symbol when jumping into a new state. This means that arbitrary events can be associated to each state of the machine. In the present case, let us simply associate the Boolean values `true` to states `UNSAFE` and `SAFE`, and the value `false` to state `ERROR`. This is done using a method called `addSymbol()`:

```
machine.addSymbol(UNSAFE, new Constant(true));
machine.addSymbol(SAFE, new Constant(true));
machine.addSymbol(ERROR, new Constant(false));
```

✎

The `addSymbol` method takes as arguments the number of a state, and a `Function` object that is expected to return the desired symbol. This function is expected to ignore its input arguments, and to have the same output arity as the Moore machine itself. In the present case, the function is a simple `Constant` that returns a `Boolean` object. We stress that the machine does not need to return a Boolean, and that any Java object can be associated to a state.

The Moore machine is now ready. It can be applied on a sequence of events, by connecting it upstream to a `QueueSource` as usual, and by pulling the events it produces.

```
QueueSource source = new QueueSource();
source.setEvents("hasNext", "next", "hasNext",
    "hasNext", "next", "next");
Connector.connect(source, machine);
Pullable p = machine.getPullableOutput();
for (int i = 0; i < 7; i++)
{
  Boolean b = (Boolean) p.pull();
  System.out.println(b);
}
```

✎

From the input events given to the source, the output of the machine should be:

```
true
true
true
true
true
false
false
```

A complete graphical representation of the chain of processors in this program would be like the diagram in Figure 5.5. Note how the transitions that were simply labelled with a method name in the original picture are replaced by a `Function` tree that checks for equality. Note also that the state numbers have been omitted, but that the output event associated to each state is shown instead.

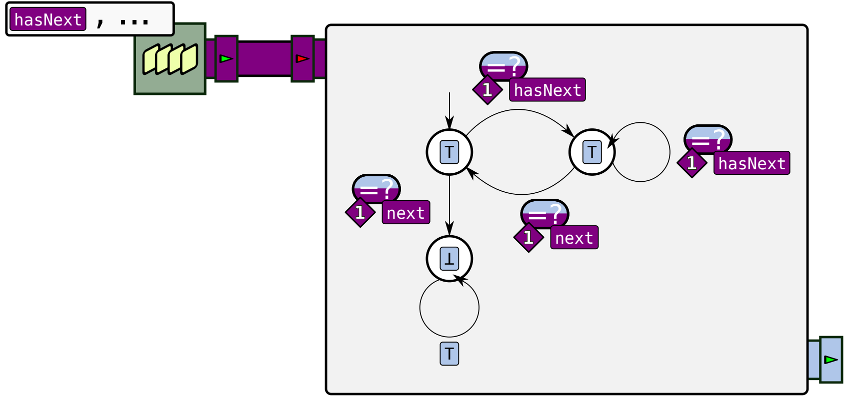


Figure 5.5: A complete representation of the `MooreMachine` example.

USING THE MACHINE'S CONTEXT

We have seen in the previous chapter how each `Processor` object carries an associative map called a `Context`. A `MooreMachine` is one example of a processor that can put this `Context` object to good use, by employing it as a storage location for local variables. These variables can be initialized by the `MooreMachine` when it is created, modified when a transition is taken, and their value can be used in the conditions that determine which transition should fire. In this respect, such variables work in a very similar way to the same kind of local variables one can find in UML state machines.

Let us modify the previous example to illustrate the use of variables. We shall tweak the state machine, and impose the (arguably bizarre) constraint that the number of calls to `hasNext()` between each call to `next()` should increase by 1 every time. Since this constraint involves counting, and we impose no upper bound on the count, it cannot be represented by a classical finite-state machine. However, this becomes possible using additional variables. The principle is to update two variables: c keeps the number of calls to `hasNext()` since the last call to `next()`, and n keeps the expected number of calls to `hasNext()` in the current "cycle". Every time `hasNext()` is called, c should be incremented. Every time `next` is called, c should be reset to zero and n should be incremented. An error occurs whenever `hasNext` is called and c is greater than n, or when `next` is called and c is not equal to n. This could be illustrated as follows:

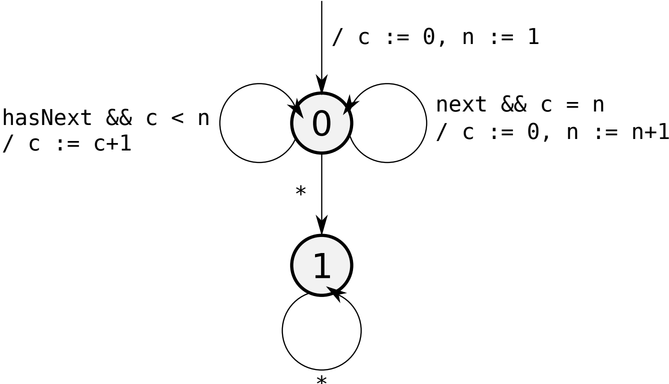


Figure 5.6: A finite-state machine representing the constraint of the second example.

This FSM looks very different as the previous one. As you can see, transitions now have conditions attached to them: these conditions are called *guards*. For example, the loop transition on the left-hand side of State 0 can be fired only if the incoming event is `hasNext` *and* the current value of local variable c is less than the current value of local variable n. In addition, transitions now also have *side effects* –that is, actions that change the processor's internal configuration other than simply moving it from one state to another. These side effects, in the figure, are separated from the guard by a slash, and consist of assignments to the local variables. When a state has multiple outgoing transitions, the $*$ is interpreted as the transition that fires when no other does.

Take two minutes to convince yourself that this "extended" Moore machine indeed corresponds to the constraint we want to enforce. Let us now attempt to create this machine in code using BeepBeep processors. The first step is to create an empty 1:1 `MooreMachine` object, and to set variables c and n to their initial values. This is done in the following code snippet. We use `Processor`'s method `setContext` to give values to two new keys, called `c` and `n`, which are added to `machine`'s `Context` object:

```
MooreMachine machine = new MooreMachine(1, 1);
machine.setContext("c", 0);
machine.setContext("n", 1);
```

✎

The next step is to define transitions, as before. Let us first consider the case

of the loop on State 0 located on the left-hand side of the figure. The guard on this transition should express the condition that:

- The current event is the string `hasNext` **and**,
- The current value of c in the processor's context is less than the current value of n.

Such a Boolean function can be created with the help of a `FunctionTree`, as is shown by the code below:

```
FunctionTree guard = new FunctionTree(And.instance,
new FunctionTree(Equals.instance,
    StreamVariable.X, new Constant("hasNext")),
new FunctionTree(Numbers.isLessThan,
    new ContextVariable("c"), new ContextVariable("n")));
```

✎

The novelty of this line of code is the use of a new type of variable, called the `ContextVariable`. When a `Function` object is evaluated inside a `Processor` (as will be the case here), a `ContextVariable` returns the value associated to the specified key in the processor's `Context` at the moment the function is evaluated. Therefore, in the present case, the function will compare the current value of c and n, every time the guard is evaluated by the Moore machine.

The transition has a guard, but also a side effect, which in this case is to increment the value of c by one. To indicate such a side effect, we need to use yet another new object, called `ContextAssignment`. The constructor of the `ContextAssignment` takes two arguments: a string that indicates the context key to modify, and a `Function` object whose return value determines the new value associated to this key. The code for creating this object looks like this:

```
ContextAssignment asg = new ContextAssignment("c",
    new FunctionTree(Numbers.addition,
        new ContextVariable("c"), new Constant(1))
    );
```

✎

In the present case, the function passed is a `FunctionTree` adding the constant 1 to the current value of c in the processor's context. Indeed, this has the effect of incrementing the processor's variable c by one.

We are now ready to add the transition to the Moore machine. This is done, as before, by using the `addTransition` method:

```
machine.addTransition(0, new FunctionTransition(
  guard, 0, asg));
```

✎

Note that, this time, the `addTransition` methods takes three arguments: the `Function` corresponding to the guard, the number of the destination state, and the `ContextAssignment` corresponding to the side effect to apply on that transition. As a matter of fact, `addTransition` accepts any number of `ContextAssignments` after its first two arguments; this makes it possible to change the value of multiple context keys in the same transition.

Once we understand these concepts, defining the other self-loop on State 0 becomes straightforward. Instead of creating separate `guard` and `asg` objects, we put everything into the same method call:

```
machine.addTransition(0, new FunctionTransition(
    new FunctionTree(And.instance,
        new FunctionTree(Equals.instance,
            StreamVariable.X, new Constant("next")),
        new FunctionTree(Equals.instance,
            new ContextVariable("c"), new ContextVariable("n"))),
    0,
    new ContextAssignment("c", new Constant(0)),
    new ContextAssignment("n",
        new FunctionTree(Numbers.addition,
            new ContextVariable("n"), new Constant(1))
        )
    ));
```

✎

Obviously, this notation tends to be a bit verbose, but in counterpart, it makes the definition of transitions and side effects very flexible.

One last comment must be made about the definition of the "star" transitions. In the previous example, we used the constant `true` as the condition for the sole star transition there was in the Moore machine. This worked, since there was no other outgoing transition on State 2. However, the order in which a Moore machine evaluates the guard on each of the outgoing transitions is non-deterministic. Setting `true` as the condition on the transition from State

0 to State 1 could lead to strange results: the FSM could move from 0 to 1 even if the condition on the other transition is true, just because it is the first one to be evaluated.

To alleviate this problem, we must use a different kind of transition, called `TransitionOtherwise`. This transition fires *if and only if* none of the other outgoing transitions from the same source state can fire. This is the object used to define the transition from State 0 to State 1, and also the self-loop on State 1:

```
machine.addTransition(0, new TransitionOtherwise(1));
machine.addTransition(1, new TransitionOtherwise(1));
```

✎

The single argument of `TransitionOtherwise`'s constructor is the destination state of the transition.

The remaining step is to associate output symbols to each state of the machine. We shall illustrate another feature of BeepBeep's `MooreMachine` object: instead of giving fixed symbols to states, we make the machine output values of their local variables. This is possible since the `addSymbol()` method requires a `Function` object; in the previous example, this function was a `Constant`. Here, a `ContextVariable` is passed, fetching the value of *c* in the processor's context, and associating it to State 0:

```
machine.addSymbol(0, new ContextVariable("c"));
```

✎

Whenever it reaches State 0, the Moore machine will query the current value of its local variable *c* and send it as its output event. This machine can be illustrated graphically as in the following figure.

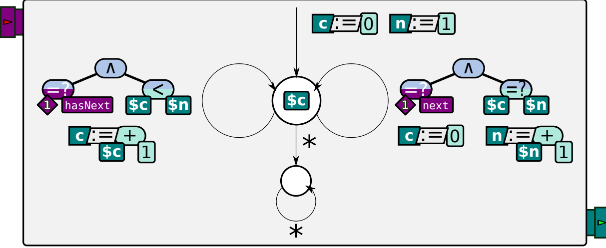


Figure 5.7: The `MooreMachine` for the second example.

We can now try this machine on a feed of events, by connecting it to a queue source as before. If the source is made of the following sequence of strings:

```
hasNext
next
hasNext
hasNext
next
next
hasNext
```

the machine should output:

```
1.0
0
1.0
2.0
0
```

Notice how the count increments, then resets to 0 upon receiving a `next` event. Moreover, upon receiving the last `next` event, the machine moves to State 1 and no longer outputs anything, as expected.

The purpose of this section is not to have an in-depth discussion on the theory of finite-state machines. The previous two examples have shown all the features of BeepBeep's `MooreMachine` processor, and highlighted its flexibility in defining guards, side effects, and associating symbols to states. In particular, our FSMs are not restricted to outputting Boolean values, and can also accept any kind of input event. A few use cases in the next chapter will further show how the `MooreMachine` can be used in various scenarios, and mixed with other BeepBeep processors.

First-order Logic and Temporal Logic

The `Booleans` utility class provides basic logical functions for combining Boolean values together; anybody who does a bit of programming has already used operators such as "and", "or" and "not". However, there is more to logic than these simple connectives. BeepBeep provides two palettes, called FOL and LTL, which extend classical logic with new operators pertaining to *first-order logic* and *linear temporal logic*, respectively. Let us examine these operators and see what they can do.

Often, we want to express the fact that a condition applies "for all objects" of some kind. For example, given a set of numbers, we could say that each of them is even; given a set of strings, we could say that each of them has at most five characters. Instead of repeating the same condition for each object, a cleaner approach consists of using what are called *quantifiers*.

In the BeepBeep world, a quantifier is a function Q that takes as parameter a String x, called the **quantification variable**, and another function f, which must have a Boolean output type. Q receives a Java `Collection` C as its input argument; for each element e in C, it evaluates f by passing it a `Context` object with the association x=e; it collects the Boolean value returned by each such call. The *universal* quantifier computes the conjunction (logical "and") of those values and returns it. In other words, a universal quantifier returns `true` if f returns `true` every time we assign to x an element in C. The *existential* quantifier rather computes the disjunction (logical "or") of those values; it returns `true` as soon as f returns `true` by replacing x by some element in C.

In BeepBeep's FOL palette, universal and existential quantifiers are implemented by two `Function` objects called `ForAll` and `Exists`. Let us illustrate the use of such quantifiers on a simple example. Consider the following piece of code:

```
Function f = new FunctionTree(Numbers.isEven, new ContextVariable("x"));
ForAll fa = new ForAll("x", f);
List<Number> nums = new ArrayList<Number>();
nums.add(2);
nums.add(6);
Object[] outputs = new Object[1];
fa.evaluate(new Object[]{nums}, outputs);
System.out.println(outputs[0]);
nums.add(3);
fa.evaluate(new Object[]{nums}, outputs);
System.out.println(outputs[0]);
```

✎

The first line creates a new `FunctionTree` `f` that simply checks if the current value of context variable x is an even number. The next line creates a `ForAll` called `fa`, with x as its quantification variable and f as its function. We then create a list containing two numbers. We proceed to evaluate `fa` on this list

(you may want to go back to the beginning of chapter 3 to recall the syntax to evaluate `Function` objects). This has for effect of evaluating f twice: the first time by setting x to 2 in the context, and the second time by setting x to 6. Both calls return `true`; the conjunction of these values is also `true`, which is the value returned by `fa` and printed at the console. This corresponds to the intuition that `fa` verifies that "all the numbers in its input set are even".

We then modify the list `nums` by appending number 7 at its tail. Re-evaluating `fa` on this list this time yields the value `false`. Three calls to f occur in the background, and the last one (corresponding to the context where x=7) returns `false`. This indeed matches the fact that not all numbers in the input set are even.

Graphically, `fa` can be represented as in the following picture:

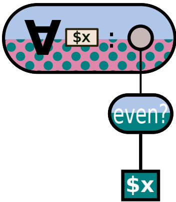


Figure 5.8: A graphical representation of the `ForAll` processor.

In this diagram, the quantified variable (in the grey box), as well as the `FunctionTree` that is used as the quantifier's function (made of the application of function `IsEven` on context variable x) can be identified. In addition, note the consistency of the colour coding:

- The quantifier accepts a collection (pink) of numbers (teal), represented by the polka dot pattern; it also returns a Boolean value (grey-blue).
- Function `IsEven` accepts a number (teal) and returns a Boolean value (grey-blue).

Quantifier `Exists` performs what is called the *dual* of the universal quantifier. It returns `true` when at least one call to the underlying function f returns `true`. In our example, replacing `ForAll` with `Exists` would check that at least one number in the input list is even.

Quantifiers can also be *nested*. That is, the underlying function given to a quantifier can itself be another quantifier. Consider a condition such as this: "all strings in a collection have the same length". It can be represented graphically

as follows:

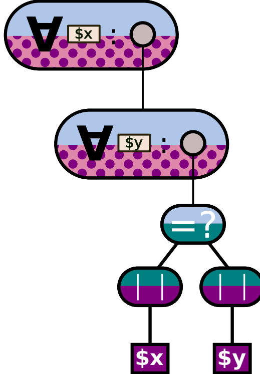


Figure 5.9: Nesting two quantifiers.

In this case, a first quantifier `fa1` creates a context object by setting the quantification variable *x* successively to each of the strings in the input collection. It then evaluates its underlying function using each context. This function turns out to be another quantifier, which is given the same input collection. Given a context and an input collection, this second quantifier (`fa2`) creates yet more context objects by taking the incoming context, and setting the quantification variable *y* successively to each of the strings in the input collection. This quantifier also evaluates an underlying function `f`, which checks the equality between the length of the string associated to context variable *x* and the length of the string associated to context variable *y*.

Programmatically, the previous figure is represented by the following program; note how `fa2` is given as the `Function` argument to the constructor of `fa1`:

```
Function f = new FunctionTree(Equals.instance,
    new FunctionTree(Size.instance, new ContextVariable("x")),
    new FunctionTree(Size.instance, new ContextVariable("y")));
ForAll fa2 = new ForAll("y", f);
ForAll fa1 = new ForAll("x", fa2);
```

✎

We can then evaluate `fa1` as in the previous example, but this time on collections of strings:

```
Set<String> strings = new HashSet<String>();
strings.add("foo");
```

```java
strings.add("bar");
Object[] outputs = new Object[1];
fa1.evaluate(new Object[]{strings}, outputs);
System.out.println(outputs[0]);
strings.add("bazz");
fa1.evaluate(new Object[]{strings}, outputs);
System.out.println(outputs[0]);
```

✎

As expected, the output of the program is:

```
true
false
```

Since quantifiers are `Function` objects like any other, there is no constraint on how quantifiers can be mixed with other `Function` objects –provided that the input and output types match, obviously. For those who know what *prenex form* is, BeepBeep functions using quantifiers do not have to be put into prenex form to be evaluated.

Each quantifier also exists in a variant which, instead of taking a set as its input, accepts an arbitrary object. When instantiated, this variant requires an extra `Function`, called the *domain function*, which is used to compute a set of elements from the input argument.

LINEAR TEMPORAL LOGIC: OPERATOR "G"

While first-order logic provides quantifiers allowing us to repeat a condition on each element of a collection, another branch of logic concentrates on ordering relationships between events in a sequence. This is called *temporal logic*, and we shall concentrate in this section on *linear temporal logic*, also called LTL.

LTL adds four new *operators* that can be used in a logical expression; these are called **G**, **F**, **X** and **U**. An LTL expression is a mix of these four operators with the traditional Boolean connectives (negation, conjunction, disjunction, implication). Let us examine the meaning of each of these operators successively. There already exists ample documentation on LTL as a logical language. In this section, we take a slightly different approach, and describe each operator by viewing it as a `Processor` on Boolean streams.

Operator **G** means "globally"; this operator is represented by a processor called (unsurprisingly) `Globally`. Its purpose is to make sure that the input stream remains `true` indefinitely.

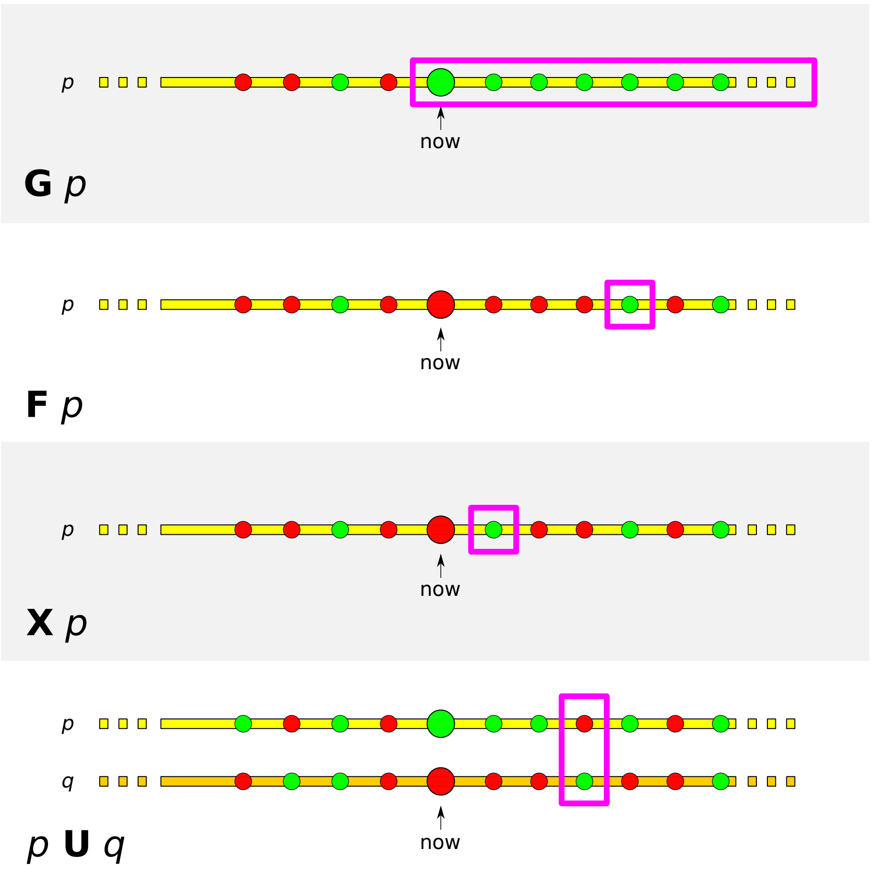


Figure 5.10: The intuitive meaning of the four LTL temporal operators.

The next figure illustrates this fact graphically. Its topmost section shows a timeline of events, represented by circles. Time flows from left to right, and the larger circle represents the current event. The colour of each circle indicates whether the input stream p is true (green) or false (red) in a particular event. As can be seen, for **G** p to return `true` on the current event, p itself must be true in the current event, but also in all subsequent events.

Consider the following code example, represented by the illustration below:

```
Globally g = new Globally();
Print print = new Print();
print.setPrefix("Output: ").setSeparator("\n");
Connector.connect(g, print);
Pushable p = g.getPushableInput();
System.out.println("Pushing true");
p.push(true);
System.out.println("Pushing true");
p.push(true);
```

✎

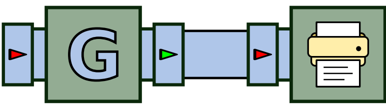


Figure 5.11: Pushing Boolean events to `Globally`.

We create a new instance of `Globally`, to which we push Boolean events – these correspond to the values of p. Before each call to `push`, we print a line at the console. However, the first lines of output of the program may look surprising:

```
Pushing true
Pushing true
```

We have pushed two events into `g`, but `g` in turn did not output anything. To understand why, we must go back to the definition we gave of operator **G**: it returns `true` on the current input event, if and only if p is true for the current event *and* all subsequent events. But how can `g` know about future events? Therefore, after receiving the first event (`true`), no definite output value for this event can be determined yet. The same reasoning applies for the second event that is pushed to `g`, which again produces no output.

Let us see what happens when we push some more events to `g`:

```
Globally g = new Globally();
Print print = new Print();
print.setPrefix("Output: ").setSeparator("\n");
Connector.connect(g, print);
Pushable p = g.getPushableInput();
System.out.println("Pushing true");
p.push(true);
```

```
System.out.println("Pushing true");
p.push(true);
```

✎

These additional lines of code produce this output:

```
Pushing false
Output: false
Output: false
Output: false
Pushing true
```

We get another surprise: pushing event `false` makes `g` push *three* output events: the constant `false` three times –but this is explainable. Upon the third call to `push()`, the stream of events e_1, e_2, e_3 received so far is the sequence `true`, `true`, `false`. Now, `g` has enough information to determine what to output for e_1: since the stream starting at this position is not made entirely of the value `true`, the corresponding output should be `false`, which explains the first output event.

However, `g` also has enough information to determine what to output for e_2 as well: for the same reason as above, the stream starting at this position is not made entirely of the value `true`; this is why `g` can afford to output a second `false` event. The third output event can also be explained: obviously, the stream starting at e_3 is not made entirely of the value `true` (as e_3 itself is false), and hence `g` can output `false` for e_3 right away.

It takes some time to get used to this principle. What must be remembered is that `Globally` delays its output for an input event until enough is known about the future to provide a definite value. As a matter of fact, `Globally` can never return `true` –how could one be sure in advance that all future events are going to be true? It can only return the value `false`, in bursts, when it receives a `false` event. As an exercise, try pushing more events to `g` in order to train your intuition.

OTHER LTL OPERATORS

Once you grasp the meaning of `Globally`, other operators are easier to understand. The LTL operator **F** is the dual of **G**, and means "eventually" (the "F" stands for "in the *future*"). If $e_1, e_2, \dots$ is a stream of Boolean events, and p is an arbitrary LTL expression, an expression of the form **F** p stipulates that p

must be true at least once at some point in the future. This is illustrated in the second section of the previous figure. As you can see, for **F** p to return true in the current event, it suffices that p be true right now, or in some event in the future. This is illustrated in the following code example:

```
Eventually e = new Eventually();
Print print = new Print();
print.setPrefix("Output: ").setSeparator("\n");
Connector.connect(e, print);
Pushable p = e.getPushableInput();
System.out.println("Pushing false");
p.push(false);
System.out.println("Pushing false");
p.push(false);
System.out.println("Pushing true");
p.push(true);
System.out.println("Pushing false");
p.push(false);
```

✎

We perform similar operations to what we did with `Globally` in the previous example. Note that the behaviour of `Eventually` can be explained in the same way, with values `true` and `false` swapped. That is, `e` outputs a burst of `true` events as soon as it receives a `true` event, and delays its output as long as it receives `false` events. Thus, the program above outputs the following lines:

```
Pushing false
Pushing false
Pushing true
Output: true
Output: true
Output: true
Pushing false
```

The third LTL operator is **X**, which means "next". It simply checks that the next event in the stream is `true`. This is illustrated in the third section of the previous figure. In BeepBeep, operator **X** is implemented by processor `Next`. Let us push events to this processor in this piece of code:

```
Next n = new Next();
Print print = new Print();
print.setPrefix("Output: ").setSeparator("\n");
Connector.connect(n, print);
```

```
Pushable p = n.getPushableInput();
System.out.println("Pushing true");
p.push(true);
System.out.println("Pushing true");
p.push(true);
System.out.println("Pushing false");
p.push(false);
System.out.println("Pushing true");
p.push(true);
```

✎

As expected, the processor does not output any event on the first call to `push()`: this output should be `true`, if and only if the *next* event in the stream is true (something we don't know about yet). As a matter of fact, the *i*-th output event is simply that of the *i*+1-th input event. Therefore, the program outputs:

```
Pushing true
Pushing true
Output: true
Pushing false
Output: false
Pushing true
Output: true
```

The last temporal operator is **U**, which stands for "until". Contrary to the previous processors, the `Until` processor takes as input two Boolean streams, which we shall call *p* and *q*. The processor checks that the event on stream *q* is `true` on some future input front, and that until then, the event on stream *p* is `true` on every input front. In other words, *p* must be true until *q* becomes true. This can be seen in the figure describing the LTL operators.

Let us interact with the `Until` processor, as in the following code snippet:

```
Until u = new Until();
Print print = new Print();
print.setPrefix("Output: ").setSeparator("\n");
Connector.connect(u, print);
Pushable p = u.getPushableInput(0);
Pushable q = u.getPushableInput(1);
System.out.println("Pushing p=true, q=false");
p.push(true); q.push(false);
System.out.println("Pushing p=true, q=false");
p.push(true); q.push(false);
```

```
System.out.println("Pushing p=true, q=true");
p.push(true); q.push(true);
System.out.println("Pushing p=false, q=false");
p.push(false); q.push(false);
```

✎

The program produces the following output:

```
Pushing p=true, q=false
Pushing p=true, q=false
Pushing p=true, q=true
Output: true
Output: true
Output: true
Pushing p=false, q=false
Output: false
```

At this point, we are more familiar with the behaviour of LTL processors. Note how `Until` delays its first output until it receives its third event front, at which point three definite output events can be produced. Indeed, starting at the first event front, we have that p has value `true` for all event fronts until q has value `true` in the third one. Hence, the first output event of the processor is `true`. The same reasoning applies when one starts at the second and third event front.

Note that `Until`, like any other synchronous processor with an arity greater than 1, waits until a complete event front is available before performing a processing step. That is, if we push events only on `p` or on `q`, processor `u` will not produce any output –but this time, this will be because it is waiting for events at matching positions in the other input stream.

NESTING LTL OPERATORS

Like quantifiers, temporal operators can be *nested*: the output of an LTL processor can be fed to the input of another one. Consider a stream of basic events called *a*, *b*, *c* and *d*, and the constraint: "between an *a* and a *d*, there cannot be a *b* immediately followed by a *c*". For example, the stream *baccbbd* satisfies this constraint, while *accbcbd* would not. In LTL parlance, this would correspond to the formula: $a \rightarrow (\neg\, (b \wedge \mathbf{X}\, c)\ \mathbf{U}\ d)$. A processor chain that checks this constraint is shown in the next figure (✎).

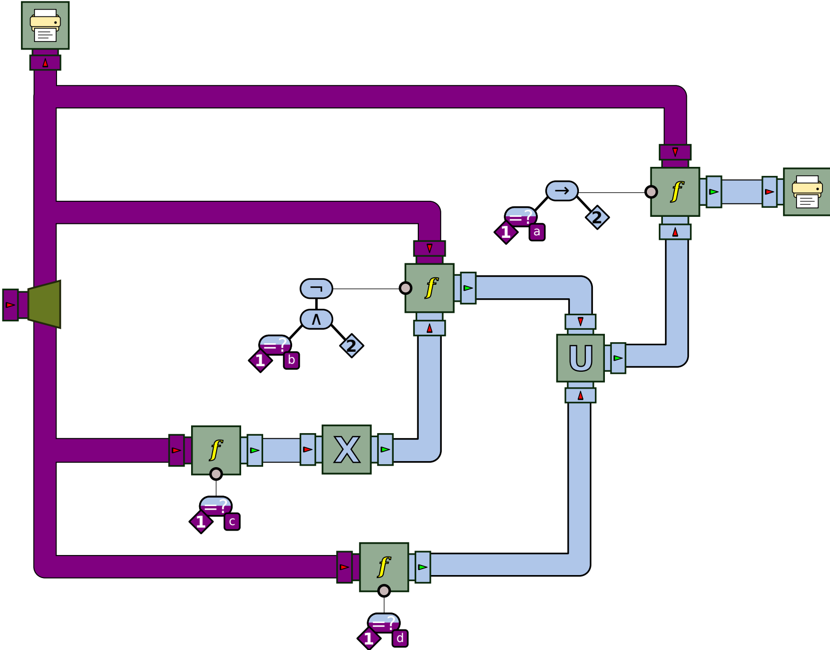


Figure 5.12: A more complex example involving multiple "nested" temporal operators.

Although this chain looks a little more complex than the previous examples, one can follow the construction of the LTL formula by reading the figure from right to left. The rightmost `ApplyFunction` implements the implication $a \rightarrow P$, where P is a Boolean trace of events created upstream. P itself corresponds to the stream coming out of the `Until` processor, which implements the sub-expression $Q\,\mathbf{U}\,d$. In turn, Q corresponds the output of the `ApplyFunction` processor that evaluates $\neg\,(b \wedge R)$, while R is the output of a processor evaluating $\mathbf{X}\,c$. One can observe that, by replacing each sub-expression in succession, the resulting LTL formula we obtain is indeed $a \rightarrow (\neg\,(b \wedge \mathbf{X}\,c)\,\mathbf{U}\,d)$.

The chain has also been fitted with two `Print` processors, to print the events that are pushed on the left, and the events that come out on the right. Pushing some events yields an output like this:

```
Pushing: c
Output: true
Pushing: d
```

```
Output: true
Pushing: a
Pushing: c
Pushing: b
Pushing: d
Output: true
Output: true
Output: true
Output: true
Pushing: f
Output: true
```

Notice how the use of an `ApplyFunctionPartial` processor on the rightmost processor has for effect of yielding an immediate verdict in some cases. The top-level expression that is ultimately evaluated is of the form $a \rightarrow P$; when the current input event is not an *a*, it is not necessary to wait for the truth value of *P* to output the value `true`. Only when the input event is an *a* must the implication "wait" before returning a value. The output of the `ApplyFunctionPartial` processor is delayed, until the processor chain taking care of the right-hand side of the implication outputs a value.

Intuitively, this processor chain can be seen as a "safeguard" mechanism. Suppose we want to prevent a program from producing a stream that violates the LTL constraint. Therefore, we would like to "monitor" an input stream, and only output its contents when we are certain that it respects the property. As long as the input stream contains events other than *a*, no constraint applies on future events. In other words, the input events, in this case, can be immediately output without fearing of violating the LTL formula.

However, when the input event is an *a*, we must make sure that no *b* is immediately followed by a *c*, and moreover, that a *d* event eventually occurs. Since we do not know what future events may come, we must delay the output of event *a* until we are sure the constraint is respected –for example by putting it into a temporary buffer. When a *d* finally comes in, we can inspect the contents of the buffer, make sure that no *b* is followed by a *c*, and, if this is the case, output the whole contents of the buffer at once. In other words, our "monitor" would act as a gatekeeper, and let the input stream get through in chunks of events that are always guaranteed to comply with the constraints.

This process is a special case of what is called *enforcement monitoring*. It turns out that in BeepBeep, creating an enforcement monitor of this kind can be done easily, by using the Boolean output of our LTL processor as the

control stream of a `Filter`. As a simple example, suppose we are monitoring a stream of operations made on a file, such as `read`, `open`, `close`, etc.). A possible constraint on this stream would be that an `open` operation must be followed later on by a `close`. In LTL, this would correspond to the expression $open \rightarrow \mathbf{F}\ close$. Consider the following processor chain (✎):

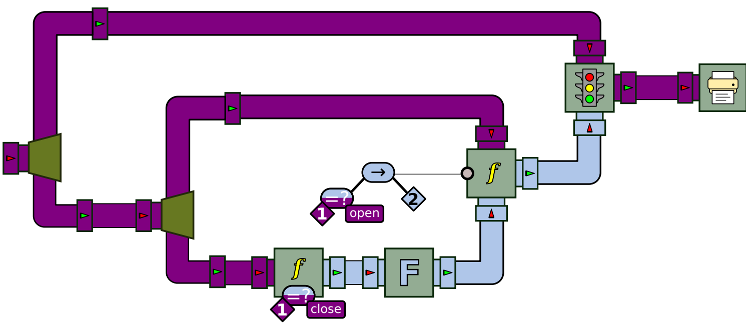


Figure 5.13: Filtering events that follow a temporal property.

The bottom part of the chain corresponds to the monitoring of the LTL formula. This output is then sent to the control pipe of a `Filter` processor, which receives on its data pipe a fork of the original stream. Pushing events on the fork produces an output like this:

```
Pushing nop
Output: nop
Pushing open
Pushing read
Pushing close
Output: open
Output: read
Output: close
```

Notice how, after pushing an `open` event, the output of the filter is buffered until a `close` is seen, after which all the buffered events are output.

There is much more to be said about monitoring in general, and LTL in particular. Although somewhat clumsy, the expression of LTL properties can be a powerful means of verifying complex ordering constraints on streams of events. The reader is referred to the appendix for more references on this topic.

Java Widgets

Up until now, none of the examples we have shown involve interaction with a user. The sample programs get their data from a fixed source, such as a text file or a predefined `QueueSource`. In the same way, apart from the basic `Print` processor, there is little in the way of displaying information to the user. The *Widgets* palette fills some of these gaps, by allowing widgets of the Java Swing graphical user interface (GUI) to be used as processors, and interact with other such objects in a chain.

In a nutshell, building a GUI in Java involves creating what are called *components*, such as windows (`JFrame`), buttons (`JButton`), sliders (`JSlider`), and defining the placement and properties of these various elements. Some of these components are sensitive to user input and other actions, and generate various kinds of objects called *events*: for example, pressing a button generates an instance of an `ActionEvent` containing information about the click (the position of the mouse, a reference to the button that was clicked, etc.). Similarly, moving the cursor of a slider generates an instance of a `ChangeEvent`.

In order for a program to react to user input, one must *register* an object implementing the `EventListener` interface (or one of its descendents). Hence, to react to a click on some `JButton` instance `b`, one would call `b.addActionListener(a)`, where `a` is an arbitrary object that implements the `ActionListener` interface. Such an object must have a method called `actionPerformed`, which receives an `ActionEvent` as its argument. It is up to the code of this method to perform the actions required by the program for this specific button click.

You may notice that the terminology used by the Swing library is very close to some core BeepBeep concepts. GUI components generate *events* at various moments in the execution of a program, depending on the interaction with the user. It would be natural to see such components as `Sources`, and to try and connect them to other BeepBeep processors. This is precisely the purpose of the *Widgets* palette, which provides an object called `ListenerSource` allowing the user to turn a Swing UI component into a BeepBeep event source.

As an example, let us create a window containing a text label and a slider widget, using simple Swing objects:

```
JFrame frame = new JFrame("My Widget Frame");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));
JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 30);
slider.setMajorTickSpacing(20);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
JLabel slider_label = new JLabel("Value", JLabel.CENTER);
slider_label.setAlignmentX(Component.CENTER_ALIGNMENT);
panel.add(slider_label);
panel.add(slider);
panel.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
frame.add(panel);
frame.pack();
frame.setVisible(true);
```

✎

The window should look like the following screenshot:

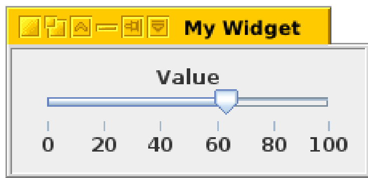


Figure 5.14: A simple window with a text label and a slider widget.

We would now like the slider to act as a BeepBeep `Source`, and send an event every time the slider's position is changed. To this end, we register a new `ListenerSource` object as a `ChangeListener` on `slider`, as follows:

```
ListenerSource ls = new ListenerSource();
slider.addChangeListener(ls);
Print print = new Print();
Connector.connect(ls, print);
```

✎

Once the `ListenerSource` is created and associated with a Swing component, it can be piped to other BeepBeep processors just like any other source. In this example, the source is simply connected to a `Print` processor, so that the events produced by the slider can be seen at the console. Once this program is started, a text line like the following should be printed at the console every time the slider is moved:

```
javax.swing.event.ChangeEvent[source=javax.swing.JSlider[,10,25,276x42,...
```

As one can see, the events generated by the slider are instances of Swing's `ChangeEvent` class. Each event contains lots of information, which could be queried using the event's accessor methods. However, most of the time, one is interested in the widget's *value*. The Widgets palette provides a BeepBeep `Function` object that extracts such value from a `ChangeEvent`. Therefore, the previous example could be modified as follows:

```
ListenerSource ls = new ListenerSource();
slider.addChangeListener(ls);
ApplyFunction gwv = new ApplyFunction(GetWidgetValue.instance);
Connector.connect(ls, gwv);
Print print = new Print();
Connector.connect(gwv, print);
```

✎

A new function called `GetWidgetValue` has been inserted between `ls` and `print`. This time, moving the slider produces a stream of numbers that are printed at the console:

```
30,28,26,25,22,...
```

In the same way that widgets can be used as event sources, some of them can also be used as sinks. The `WidgetSink` is a 0:1 processor that is instantiated by giving it a Swing widget. When it receives an event, it sets the widget's state according to the event's content. This may mean different things, according to the widget and the event's type. For example, if the widget is a text label (`JLabel`) and the input event is a number or a string, the `WidgetSink` will use the event to set the label's text. However, if the event is an instance of Swing's `ImageIcon`, the sink will use it to set the label's background.

In the case of a slider, the `WidgetSink` expects a numerical value, and uses it to change the slider's position. Using the same `JFrame` as the previous example, we can therefore write a piece of code like the following:

```
WidgetSink ws = new WidgetSink(slider);
Pushable p = ws.getPushableInput();
for (int i = 10; i <= 100; i+= 10)
{
  p.push(i);
  Thread.sleep(1000);
}
```

✎

Notice how, this time, the program pushes events into the `WidgetSink` associated to the slider. By running this program, you should see the slider jumping from value 10 to 100, in increments of 10, every second.

The palette also includes another `Function` object, called `ToImageIcon`, which converts an array of bytes into a Swing `ImageIcon` object. It can be useful to take the output of a processor that produces an image (such as a PNG or JPEG bitmap), and to display it inside a Swing component. The next section will show an example that uses this function to display a plot and dynamically update it.

Plots

One interesting purpose of processing event streams is to produce visualizations of their content –that is, to derive plots from data extracted from events. BeepBeep's `plots` palette provides a few processors to easily generate dynamic plots.

Internally, the palette makes use of the MTNP library (MTNP stands for "Manipulate Tables N'Plots"), which itself relies on either GnuPlot or GRAL to generate the plots. The technique can be summarized as follows:

1. Event streams are used to update the contents of a structure called a **table**.
2. The contents of this table can be processed by applying a series of **transformations**.
3. The resulting table is given as the source for a **plot** object.
4. The plot is asked to produce a picture from the contents of the table.

Let us start with the table. This data structure is represented by the `Table` class of the MTNP library. A table is simply a collection of *entries*, with each entry containing a fixed number of key-value pairs. An entry therefore corresponds to a "line" of a table, and each key corresponds to one of its "columns".

A table can be created from the contents of event streams with the use of BeepBeep's `UpdateTable` processor. This processor exists in two flavours: `UpdateTableStream` takes multiple input streams, one for the value of each column; `UpdateTableArray` takes a single stream, which must be made of

arrays of values or `TableEntry` objects. Both processors perform the same action: they update an underlying `Table` object, adding one new entry to the table for each event front they receive.

The following code sample illustrates the operation of `UpdateTableStream`:

```
QueueSource src1 = new QueueSource().setEvents(1, 2, 3, 4, 5);
QueueSource src2 = new QueueSource().setEvents(2, 3, 5, 7, 4);
UpdateTable table = new UpdateTableStream("x", "y");
Connector.connect(src1, OUTPUT, table, TOP);
Connector.connect(src2, OUTPUT, table, BOTTOM);
Pump pump = new Pump();
Print print = new Print().setSeparator("\n");
Connector.connect(table, pump, print);
pump.turn(4);
```

✎

Two sources of numbers are created and are piped into an `UpdateTableStream` processor. This processor is instantiated by giving two strings to its constructor. These strings correspond to the names of the columns in the table; the number of strings also determines the input arity of the processor. The first input pipe will receive values for column "x", while the second input pipe will receive values for column "y". A pump and a print processor are connected to the output of the table updater.

After a single activation of the pump, the program should print:

```
x,y
---
1,2
```

Values 1 and 2 have been extracted from `src1` and `src2`, respectively. From this event front, the `UpdateTableStream` processor creates one table entry with x=1 and y=2, adds it to its table and outputs the table. This is relayed to the `Print` processor which displays its content. The output of the program shows that upon each new event front, one new entry in the table is added; therefore, after activating the pump four times, the last output is:

```
x,y
---
1,2
2,3
3,5
```

```
4,7
```

The next part of the process is to draw plots from the content of a table. This is the job of the `DrawPlot` processor. This processor is instantiated by being given an empty `Plot` object from the MTNP library. When it receives a `Table` from its input pipe, it passes it to the plot, and calls the plot's `render` method to create an image from it. Therefore, the output events of `DrawPlot` are *pictures* –or more precisely, arrays of bytes corresponding to a bitmap in some image format (PNG by default).

As a more elaborate example, take a look at the following program.

```
QueueSource one = new QueueSource().setEvents(1);
Pump pump = new Pump(1000);
RandomTwoD random = new RandomTwoD();
Connector.connect(one, pump, random);
UpdateTable table = new UpdateTableStream("x", "y");
Connector.connect(random, TOP, table, TOP);
Connector.connect(random, BOTTOM, table, BOTTOM);
DrawPlot draw = new DrawPlot(new Scatterplot());
Connector.connect(table, draw);
BitmapJFrame window = new BitmapJFrame();
Connector.connect(draw, window);
window.start();
System.out.println("Displaying plot. Press Ctrl+C "
    + "or close the window to end.");
Thread th = new Thread(pump);
th.start();
```

✎

A stream of (x,y) pairs is first created, with x an incrementing integer, and y a randomly selected number. This is done through a special-purpose processor that is called `RandomTwoD`. It actually is a `GroupProcessor` that internally forks an input of stream of numbers. The first fork is left as is and becomes the first output stream. The second fork is sent through a `RandomMutator` (which converts any input into a random integer) and becomes the second output stream. The resulting x-stream and y-stream are then pushed into an `UpdateTableStream` processor. This creates a processor with two input streams, one for the "x" values, and the other for the "y" values. Each pair of values from the x and y streams is used to append a new line to the (initially empty) table. The two outputs of the random processor are then connected to these two inputs.

The next step is to create a plot out of the table's content. The `DrawPlot` processor receives a Table and sends it to a `Plot` object from the MTNP library. In the current case, we want to create a scatterplot from the table's contents; therefore, we pass an empty `Scatterplot` object. As previously mentioned, each event coming out of the `DrawPlot` processor is an array of bytes corresponding to a bitmap image. To display that image, we use yet another special processor called `BitmapJFrame`. This processor is a sink that manages a `JFrame` window; when it receives an input event (i.e. an array of bytes), it turns that array into an image and displays it inside the window.

Graphically, this chain of processors can be illustrated as follows:

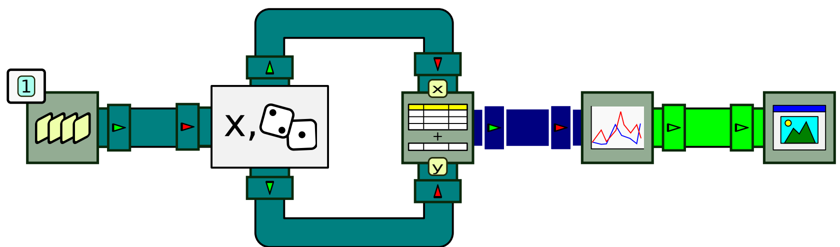


Figure 5.15: Producing a scatterplot from a source of random values.

This drawing introduces a few new boxes. The one at the far right is the `BitmapJFrame`; its input pipe is coloured in light green, which represents byte arrays. The box at its left is the `DrawPlot` processor. This processor is depicted with an icon indicating the type of plot that must be produced (here, a two-dimensional scatterplot). Still more to the left is the `TableUpdateStream` processor. Next to each of its input pipes, a label indicates the name of the column that will be populated by values from that stream. The output pipe of this processor is coloured in dark blue, representing `Table` objects.

A window containing a plot, whose contents are updated once every second (due to the action of an intermediate `Pump` object) will appear by running this program.The window should look like Figure 5.16.

Each new table entry increments the value of x by one; the value of y is randomly chosen. The end result is a dynamic plot created from event streams; the whole chain, from source to the actual bitmaps being displayed, amounts to only 12 lines of code. Obviously, sending the images into a bland `JFrame` is only done for the sake of providing an example. In a real-world situation, one would be more likely to divert the stream of byte arrays somewhere else, such

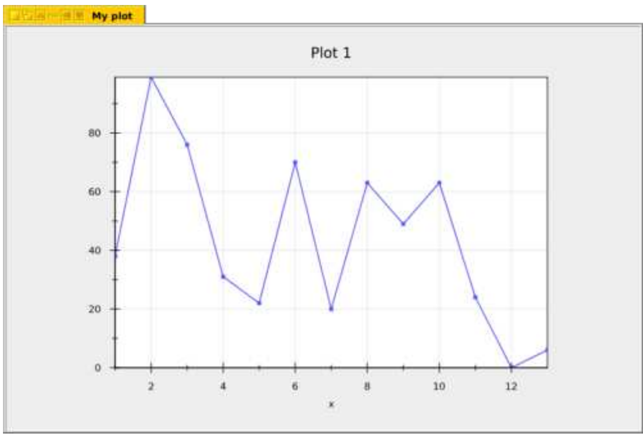


Figure 5.16: The window produced by the `BitmapJFrame` processor.

as a file, or as a component of the user interface of some other software.

Besides scatterplots, any other plot type supported by the MTNP library can be sent to `DrawPlot`'s constructor. It includes histograms, pie charts, heat maps, and so on. The only important point is that each plot is expected to receive tables structured in a particular way; for instance, a heat map requires a table with three columns, corresponding to the x-coordinate, y-coordinate, and "temperature" value, in this specific order. The upstream processor chain is responsible to produce a `Table` object with the appropriate structure.

Plots can also be customized by applying modifications to the `Plot` object sent to `DrawPlot`. For example, to set a custom title, one simply has to send an instance of `Scatterplot` whose title has been changed using its `setTitle` method:

```
Scatterplot plot = new Scatterplot();
plot.setTitle("Some title");
DrawPlot dp = new DrawPlot(plot);
```

Since the `plots` palette is a simple wrapper around MTNP objects, the reader is referred to this library's online documentation for complete details about plots, tables, and table transformations.

Signal Processing

The input of a processor chain may be a stream of numerical values obtained from physical measurements, such as temperature or power sensors. In those cases, it may be desirable to transform this "raw" signal into a higher-level stream of values, on which some preliminary clean-up has been performed. The *Signal* palette provides processors suitable for some basic signal processing tasks, such as finding peaks, plateaus, etc.

To illustrate the operation of *Signal*'s various processors, we shall first generate a stream of values representing a "signal". To this end, we use the following processor chain (✎):

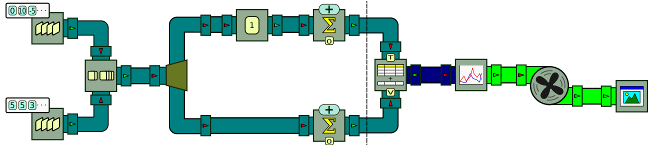


Figure 5.17: Producing a numerical signal that varies with time.

This example is one of the firsts using the `VariableStutter` processor. In the previous processor chain, it is represented by the box connected into the `Fork`. Its first input (top) is a stream of values, while its second input (bottom) indicates how many times each value should be repeated in the output. With the numbers contained in the two sources, the processor is expected to output the value 0 five times, followed by the value 10 five times, and so on.

This stream is then forked in two copies. The topmost path should be familiar to the reader, and creates a simple counter producing the output values 1, 2, 3, ...; these values will act as a clock tick "T". The bottom path cumulates the values of the forked input stream. This will produce an output signal "V" whose values move up and down upon each clock tick, from a relative amount defined by the number in the input stream.

This chain of seven processors gives us a crude way of producing a numerical signal whose behaviour is somewhat controlled by the contents of the two `QueueSources`. In our example, in order to better see the end result, the pairs of values from "T" and "V" are sent into an `UpdateTableStream` processor, transformed into a plot and displayed in a window. Running this program should show a plot like the following:

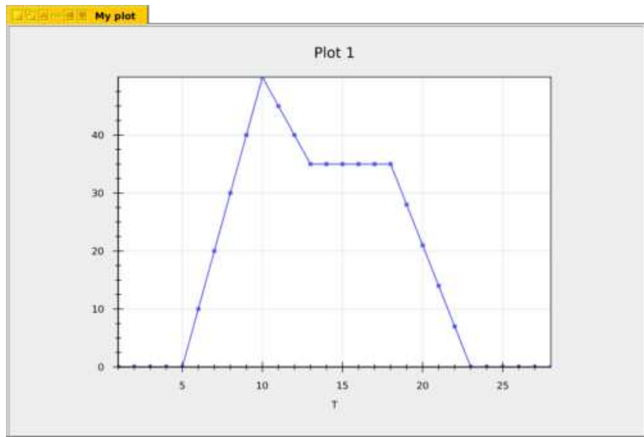


Figure 5.18: Plotting the numerical signal produced by the previous chain of processors.

As an exercise, try changing the contents of the two sources to see the effect they have on the resulting plot.

We shall use this simple "signal generator" to illustrate the operation of various processors of the *Signal* palette. To simplify both the code and the diagrams, we shall put the previous processors into two `GroupProcessors`: the first half (up to the vertical dotted line) into a group called `GenerateSignal`, and the second into a group called `PlotSignal`. Using these two groups, the processor chain shown in the last diagram can be simplified into the following one:

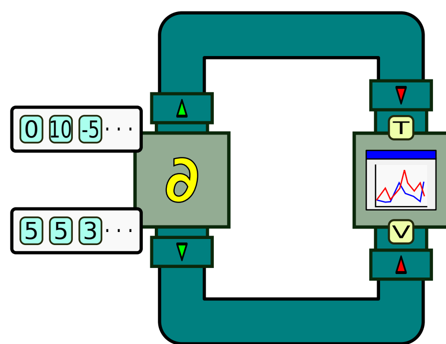


Figure 5.19: Producing a numerical signal that varies with time (grouped version).

As one can see, the source box is parameterized by the contents of the two

input queues, while the sink box is parameterized by the number and the names of each stream of numbers it receives. We use the “delta” letter in the source, since the two input queues of this processor encode a discrete form of the first derivative of the input signal to generate. The reader is encouraged to have a look at the code of `GenerateSignal` (✎) and `PlotSignal` (✎), in the examples repository, to better understand how these two groups are implemented.

A first useful processor of the *Signal* palette is used to find **peaks** in an input stream. A peak is informally defined as an abrupt increase in the values of the signal over a short number of values (or *samples*). One possible way of looking for a peak is to use a sliding window of a few samples, and to identify local maxima in this window. The `PeakFinderLocalMaximum` processor does exactly that. Consider the following code snippet:

```
GenerateSignal gs = new GenerateSignal(
    new Object[] {0, 20, -10, 0, -7, 0},
    new Object[] {5, 5, 3, 5, 5, 5});
Fork fork = new Fork(2);
Connector.connect(gs, 1, fork, 0);
PeakFinderLocalMaximum peak = new PeakFinderLocalMaximum(5);
Connector.connect(fork, 1, peak, 0);
PlotSignal ps = new PlotSignal("T", "V", "P");
Connector.connect(gs, 0, ps, 0);
Connector.connect(fork, 0, ps, 1);
Connector.connect(peak, 0, ps, 2);
ps.start();
```

✎

Using the graphical conventions we just established, this chain of processors can be represented as in Figure 5.20.

In this program, the “signal” stream produced by the delta box is forked in two parts. One of them goes directly to the `PlotSignal` processor as before. The other is first passed through the `PeakFinderLocalMaximum` processor. This processor is parameterized by the length of the window, which, in this case, is of five events. This processed signal is also fed to the `PlotSignal` box, and given the name P. Therefore, the resulting plot will be made of *two* lines: one joining points from the pairs of numbers (T,V), and another joining points from the pairs (T,P). This makes it possible to visualize the effect of the `PeakFinderLocalMaximum` processor on the same plot as the original signal.

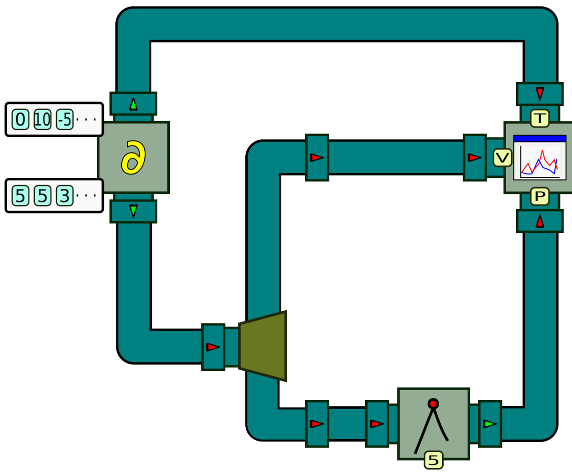


Figure 5.20: Finding peaks in a numerical signal.

The result should be a graph like the following:

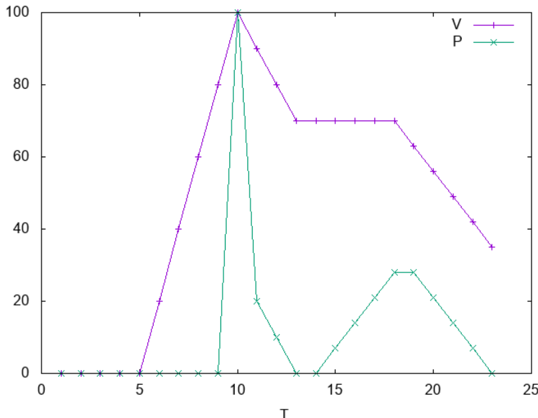


Figure 5.21: The original signal (V) and the detected local maxima (P).

As one can see, the processor outputs the value 0 if the current input event is not considered as a peak; otherwise, it outputs the height of that peak. The definition of what constitutes a peak varies, depending on the underlying algorithm that is being used; in the present case, any local maximum that exits the sliding window is considered as a peak. The current version of the *Signal* palette also provides another processor, the `PeakFinderTravelRise`,

which uses a different algorithm for detecting peaks.

The `PlateauFinder` processor identifies "plateaux" in an input signal; a *plateau* is a sequence of successive values that lie within the same (narrow) range. In the previous program, we can replace the peak processor with `PlateauFinder` and plot the results again (✎). This will produce the following plot:

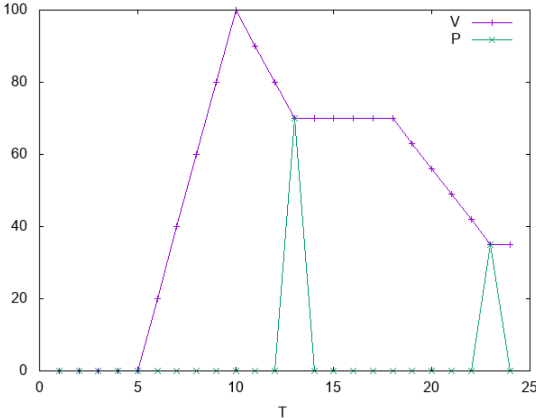


Figure 5.22: The original signal (V) and the detected plateaux (P).

We can observe that the processor outputs the value 0 when no plateau is detected; otherwise, it outputs the height of the plateau at the position of the event that corresponds to the start of a plateau. Obviously, for this processor to detect a plateau, a delay in the output is required: the start of a plateau can only be ascertained until a few events later, when enough values in the same interval have been observed. This interval is called the *window width*, and it can be configured by passing this width to the object's constructor.

Let us now change our input signal by changing its envelope and adding some random noise to its values. The processor chain to generate the signal is modified to look as in Figure 5.23.

The main difference lies in the presence of a new fork on the bottom branch. The output signal from the `VariableStutter` processor is forked one more time; on the first path (top), the signal is sent into a processor called `Randomize`; this processor turns any input event into a floating-point number, which is randomly selected from an predefined interval. In the current example, the interval is from -2 to 2, as indicated by the two numbers on the processor's

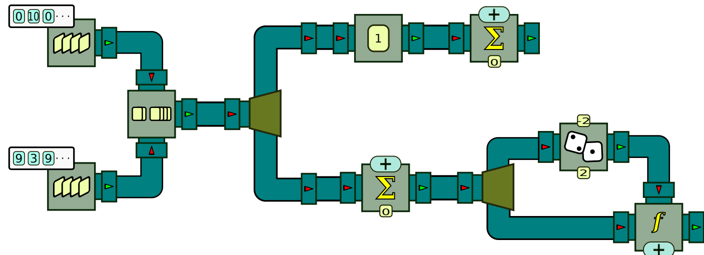


Figure 5.23: Generating a signal and adding some noise.

box. This stream of random numbers is then added to the original signal (second path). This will result in a “jagged” output signal, with the amount of variation being parameterized by the interval set on `Randomize`. The code for this modified signal generator can be found in the example repository (✎).

Equipped with this new signal generator, we can illustrate a few more processors from the *Signal* palette. The first is called `Threshold`. Its task is to flatten to zero any input number whose absolute value lies below a predefined threshold, and to let the other numbers through. An example program showing the use of this processor produces the following plot (✎):

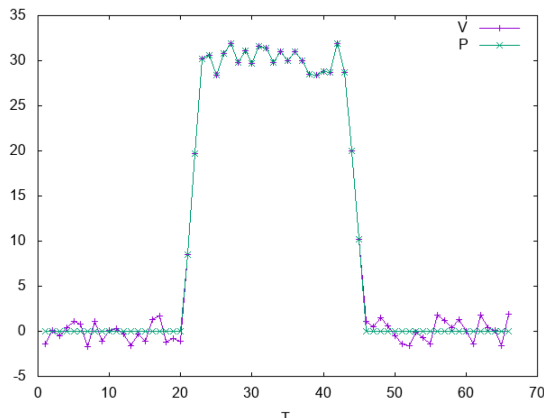


Figure 5.24: The original signal (V) and the signal after the application of the `Threshold` processor (P).

Here, the threshold has been set to 4, meaning that all values lying between -4 and 4 will be turned into 0 in the output signal. Notice how this has for effect of partly “de-noising” the input, by removing the small signal variations

around the x-axis.

Like `PlateauFinder`, the `Persist` processor also operates on a window of width k; it returns the maximum value of the window. This has for effect of "persisting" high values in a signal for some time after they occur, in a way similar to some graphic equalizers used in music software. The examples repository contains a program that illustrates the use of `Persist`; it produces a plot like the following (✎):

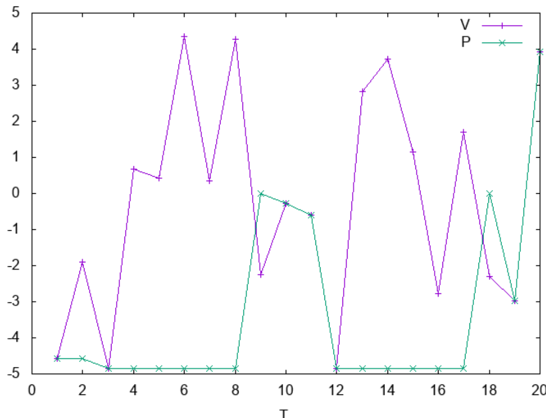


Figure 5.25: The original signal (V) and the signal after the application of the `Persist` processor (P).

As one can see, the high values in a window remain in the output for a number of events after they occur, when no higher value is observed in the sliding window.

The last processor contained in the *Signal* palette is called `Limit`. Instead of preserving high values, as is the case for `Persist`, this processor rather restricts the amount of non-zero events that can be output in a certain interval of time. The processor is instantitated with a window width k; when it receives a non-zero event, it outputs it, but will then turn into 0 the next k-1 events, regardless of whether they are zero or not. This is shown by Figure 5.26, which applies the `Limit` processor to an input signal with a window width of 4 (✎):

The *Signal* palette is still under development; it currently only provides basic processors for manipulating raw streams of numerical values. In particular, all the processors contained in the palette operate on the *time* domain; the addition of processors working on the *frequency* domain (such as Fourier

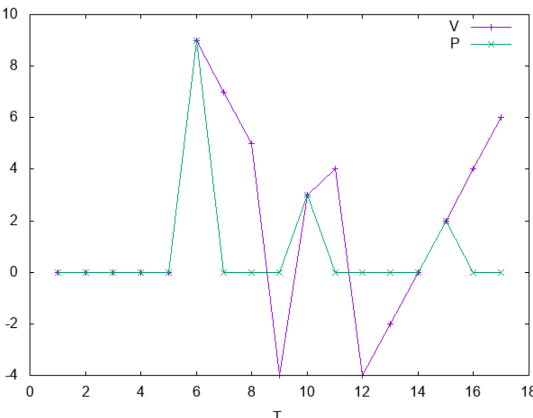


Figure 5.26: The original signal (V) and the signal after the application of the `Limit` processor (P).

transforms) is planned in future development steps. Nevertheless, the next chapter will show an example of an actual use case that uses processors from the *Signal* palette in its current state.

Networking

In the last chapter, we saw the `HttpGet` processor that fetches a character string remotely through an HTTP GET request. The `http` palette provides additional processors that making it possible to push and pull events across a network using HTTP. By splitting a processor chain on two machines and having both ends use HTTP to send and receive events, we are achieving what amounts to a rudimentary form of **distributed computing**.

In line with BeepBeep's general design principles, these functionalities are accessible through just a few lines of code. More precisely, send and receive operations are taken care of by two "gateway" processors, respectively called the `HttpUpstreamGateway` and the `HttpDownstreamGateway`.

The `HttpUpstreamGateway` is a sink processor that works in push mode only. It receives character strings, and is instructed to send them over the network as the payload of an HTTP request directed to a specific address. Thus, when instantiating the gateway, we must specify the URL where the request is ex-

pected to be sent.

The HttpDownstreamGateway works in reverse. It continually listens for incoming HTTP requests on a specific TCP port; when a request matches the URL that was specified to its constructor, its contents are pushed to its output pipe in the form of a character string.

The following program shows a simple use of these two gateways.

```
HttpUpstreamGateway up_gateway =
    new HttpUpstreamGateway("http://localhost:12144/push");
HttpDownstreamGateway dn_gateway =
    new HttpDownstreamGateway(12144, "/push", Method.POST);
Print print = new Print();
Connector.connect(dn_gateway, print);
up_gateway.start();
dn_gateway.start();
Pushable p = up_gateway.getPushableInput();
p.push("foo");
Thread.sleep(1000);
p.push("bar");
up_gateway.stop();
dn_gateway.stop();
```

✎

First, an upstream gateway is created, and is asked to send requests at a specific URL on the local machine (localhost) on TCP port 12144 (this number is chosen arbitrarily; any unused port number would work). Additionally, a "page" is specified where the gateway should push to; in this case, it is named push, but this could be any character string. The same thing is done with a downstream gateway, which is instructed to listen to port 12144, watch for URLs with the string /push (this is the same page name that was given to the upstream gateway), and to answer only to HTTP requests that use method POST. This gateway is connected to a Print processor to show what it receives on the console.

Both upstream and downstream gateways must be started in order to work; method start takes care of initializing the objects required for the network connection. Ideally, the gateways should also be stopped at the end of the program. Other than that, they work like any normal source and sink. Strings are pushed to up_gateway; after the call to push, the standard output should display the contents of that string.

So far, it seems that events were merely pushed and printed at the console. What actually happened is a bit more complex: note how the upstream and the downstream gateways have never been linked using a call to `connect`. Rather, an HTTP request was used to pass the strings around. Therefore, this program is structured as if there were two "machines" running in parallel; Machine A pushes strings through HTTP requests, and Machine B receives and prints them. This could be illustrated as follows:

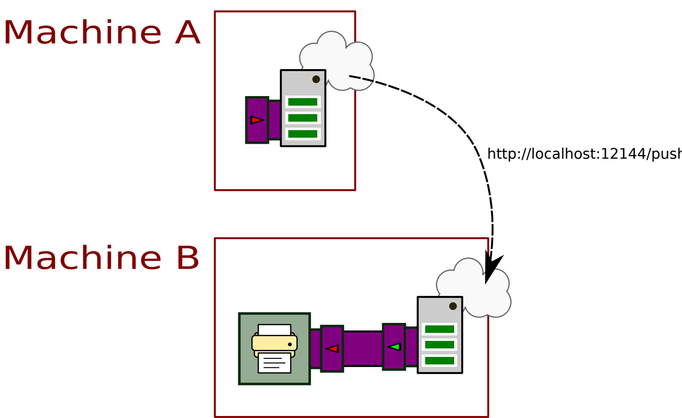


Figure 5.27: Using gateways to send events through HTTP.

It just happens that in this simple program, the HTTP requests are sent to `localhost`; therefore, they never leave the computer. However, the whole process would be identical if the character strings were sent over an actual network: `localhost` would simply be replaced by the IP address of some other computer.

SERIALIZATION

Sending character strings over a network is an arguably simple task. Very often, the events that are exchanged between processors are more complex: what if a set, a list, or some other complex object having member fields needs to be transmitted? The HTTP gateways always expect character strings, both for sending and for receiving.

A first solution would be to create a custom `Function` object that takes care of converting the object we want to send into a character string, and another one to do the process in reverse, and transform a string back into an object

with identical contents. This process is called **serialization**. However, doing so manually suggests that for every different type of object, a different pair of functions must be created to convert them to and from strings. Moreover, this process can soon become complicated. Take the following class:

```
public class CompoundObject
{
    int a;
    String b;
    CompoundObject c;
}
```

This class has for member fields an integer, a string and yet another instance of `CompoundObject`. Converting such an object into a character string requires adding delimiters to separate the int and String fields, and yet more delimiters to represent the contents of the inner `CompoundObject` –and so on recursively.

Luckily, *serialization libraries* can automate part of the serialization process. BeepBeep has a palette called `serialization` whose purpose is to provide a few functions to serialize generic objects; under the hood, it uses the Azrael serialization library. The palette defines two main `Function` objects:

- The `JsonSerializeString` function converts an object into a character string in the **JSON** format.
- The `JsonDeserializeString` function works in reverse: it takes a JSON string and recreates an object from its contents.

These two functions can be sent to an `ApplyFunction` processor, and be used as a preprocessing step before and after passing strings to the HTTP gateways.

Let us add a constructor and a `toString` method to our `CompoundObject` class:

```
public CompoundObject(int a, String b, CompoundObject c)
{
    super();
    this.a = a;
    this.b = b;
    this.c = c;
}
@Override
public String toString()
{
    return "a = " + a + ", b = " + b + ", c = (" + c + ")";
}
```

✎

Now, consider the following code example, which is a slightly modified version of the first program:

```
ApplyFunction serialize = new ApplyFunction(new JsonSerializeString());
HttpUpstreamGateway up_gateway =
    new HttpUpstreamGateway("http://localhost:12144/push");
HttpDownstreamGateway dn_gateway =
    new HttpDownstreamGateway(12144, "/push", Method.POST);
ApplyFunction deserialize = new ApplyFunction(
        new JsonDeserializeString<CompoundObject>(
            CompoundObject.class));
Print print = new Print();
Connector.connect(serialize, up_gateway);
Connector.connect(dn_gateway, deserialize);
Connector.connect(deserialize, print);
up_gateway.start();
dn_gateway.start();
```

✎

The main difference is that a processor applying `JsonSerializeString` has been inserted before the upstream gateway, and another processor applying `JsonDeserializeString` has been inserted after the downstream gateway; the rest is identical. The serialization/deserialization functions must be passed the class of the objects to be manipulated. Here, we decide to use instances of `CompoundObjects`, as defined earlier. Graphically, our processor chain becomes:

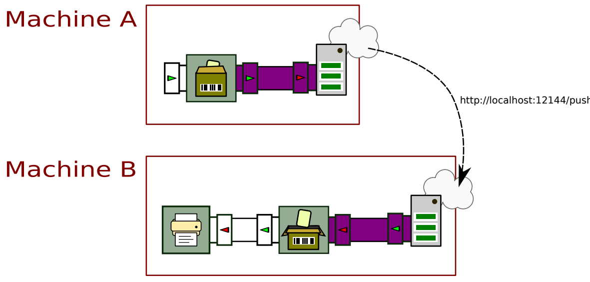


Figure 5.28: Serializing objects before using HTTP gateways.

Note the pictogram used to illustrate the serialization processor: the picture represents an event that is "packed" into a box with a bar code, representing its

serialized form. The deserialization processor conversely represents an event that is "unpacked" from a box with a bar code. Although these processors are actually plain `ApplyFunction` processors, we represent them with these special pictograms to improve the legibility of the drawings.

We can now push `CompoundObjects` through the serializer, as is shown in the following instructions:

```
Pushable p = serialize.getPushableInput();
p.push(new CompoundObject(0, "foo", null));
Thread.sleep(1000);
p.push(new CompoundObject(0, "foo", new CompoundObject(6, "z", null)));
```

✎

The expected output of the program should be:

```
a=0, b=foo, c=(null)
a=0, b=foo, c=(a=6, b=z, c=(null))
```

It is not very surprising, but one must remember all the tasks happening in the background:

- The object was converted into a JSON string.
- The string was sent over the network through an HTTP request...
- Converted back into a `CompoundObject` identical to the original...
- And pushed downstream to be handled by the rest of the processors as usual.

The entire process requires about 10 lines of code only.

ALL TOGETHER NOW: DISTRIBUTED TWIN PRIMES

As we mentioned earlier, the use of HTTP gateways provides a simple way to distribute computation over multiple computers. As a matter of fact, any chain of processors can be "split" into parts, with the loose ends attached to upstream and downstream gateways.

As a slightly more involved example, let us compute twin primes by splitting the process across two machines over a network. Twin primes are pairs of numbers p and p+2 such that both are prime. For instance, (3,5), (11,13) and (17,19) are three such pairs. The twin prime conjecture asserts that there exists an infinity of such pairs.

The program will be composed of two machines, called A and B. Machine A will be programmed to check if each odd number 3, 5, 7, etc. is prime. If so, it will send the number n to Machine B, which will then check if n+2 is prime. If it is so, Machine B will print the values of n and n+2. Checking if a number is prime is an operation that becomes very long for large integers (especially with the algorithm we use here). By verifying n and n+2 on two separate machines, the whole processor chain can actually run two primality checks at the same time.

Since computations will be done over very large numbers, the program will use Java's `BigInteger` class instead of the usual `int`s or `long`s. Furthermore, it is assumed that there exists a function object called `IsPrime`, whose purpose is to check whether a big integer is a prime number. (The code for `IsPrime` can be found in BeepBeep's code example repository.) Let us start with the program for Machine A.

```
String push_url = "http://localhost:12312/bigprime";
QueueSource source = new QueueSource().addEvent(new BigInteger("2"));
Pump pump = new Pump(500);
Connector.connect(source, pump);
Cumulate counter = new Cumulate(
    new CumulativeFunction<BigInteger>(BigIntegerAdd.instance));
Connector.connect(pump, counter);
Fork fork1 = new Fork(2);
Connector.connect(counter, fork1);
ApplyFunction prime_check = new ApplyFunction(IsPrime.instance);
Connector.connect(fork1, LEFT, prime_check, INPUT);
Filter filter = new Filter();
Connector.connect(fork1, RIGHT, filter, LEFT);
Connector.connect(prime_check, OUTPUT, filter, RIGHT);
Fork fork2 = new Fork(2);
Connector.connect(filter, fork2);
Print print = new Print();
Connector.connect(fork2, LEFT, print, INPUT);
ApplyFunction int_to_string =
    new ApplyFunction(BigIntegerToString.instance);
HttpUpstreamGateway up_gateway = new HttpUpstreamGateway(push_url);
Connector.connect(fork2, RIGHT, int_to_string, INPUT);
Connector.connect(int_to_string, up_gateway);
```

✎

First, the URL where prime numbers will be pushed downstream is specified.

The first processor is a source that will push the BigInteger "2" repeatedly. The second processor is a simple counter. It is fed with the BigInteger "2" repeatedly, and it returns the cumulative sum of those "2" as its output. Since the start value of BigIntegerAdd is one, the resulting sequence is made of all odd numbers. The events output from the counter are duplicated along two paths. Along the first path, the numbers are checked for primality. Along the second path, a filter uses the primality verdict as the filtering condition. What results from the filter are only prime numbers. The output of the filter is then forked, so that what results from it can be printed. `BigIntegers` are converted to strings, and pushed across the network to Machine B using the `HttpUpstreamGateway`.

Graphically, the chain of processors for Machine A can be represented as follows:

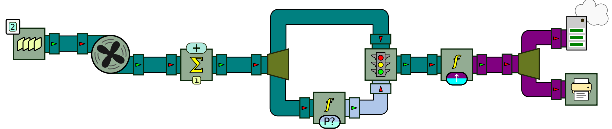


Figure 5.29: The chain of processors for Machine A.

Let us now move to Machine B. Only the processor chain is shown below:

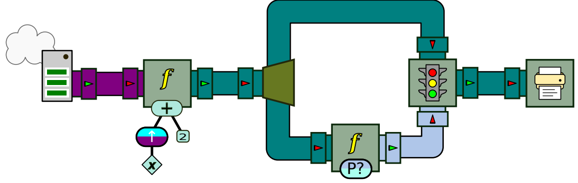


Figure 5.30: The chain of processors for Machine B.

An `HttpDownstreamGateway` is first created to receive strings from Machine A. The next step is to convert the string received from the gateway back into a `BigInteger`. This number is then incremented by 2 using the addition function for `BigIntegers`. The rest of the chain is similar to Machine A: a filter is used to only let prime numbers through, and these numbers are then printed at the console.

All in all, in this example less than 50 lines of code were written. This results in a distributed, streaming algorithm for finding twin prime numbers. Note that

this chain of processors is only meant to illustrate a possible use of the HTTP gateways. As such, it is not a very efficient way to find twin primes: when n and n+2 are both prime, three primality checks will be conducted: Machine A will first discover that n is prime, which will trigger Machine B to check if n+2 also is. However, since Machine A checks all odd numbers, it will also check for n+2 in its next computation step.

As a side note, one can see that Machine B's program depends on the numbers sent by Machine A. Therefore, if Machine A is stopped and restarted, Machine B will restart the sequence of twin primes from the beginning.

JSON and XML Parsing

We have already seen how BeepBeep can process input streams such as CSV text files, and break each line of these files into a structured object called a *tuple*. Other BeepBeep palettes can also process input data in a variety of other formats. In this section, we elaborate on two such formats, called JSON and XML.

JSON PARSING

The serialization example in the previous section alluded to a particular way of formatting information using a notation called **JSON**. This acronym stands for *JavaScript Object Notation*, as it was first used in the JavaScript programming language to represent "semi-structured" data. A JSON object is a textual document such as this:

```
{
  "a" : 0,
  "b" : [1, 2, 3],
  "c" : {
    "d" : true,
    "e" : [
      {"f": "foo"},
      {"f": "bar"}
    ]
  }
}
```

The top-level object is delimited by the outermost pair of braces. It is an associative map between keys (the character strings "a", "b", …) and values (on the right-hand side of the colon). A value can be:

- a primitive type such as a number (the value of "a"), a Boolean (the value of "d") or a character string (the value of "f");
- a list of primitive types (the value of "b") or of other JSON objects (the value of "e"); lists are denoted by square brackets;
- another JSON object (the value of "c").

As you can see, this notation allows an arbitrary nesting of objects within lists or other objects, which makes it both easy to read and quite versatile. An increasing number of applications uses this lightweight format to exchange, and sometimes even store data. We also learned in the previous section that JSON is one of the formats used by the Azrael library to serialize the contents of a Java object.

A complete tutorial on JSON is out of the scope of this section. However, it is interesting to know that a BeepBeep palette exists to parse and query JSON objects. The parsing is done with a function called `ParseJson`: it receives a character string as input, and produces an instance of an object called `JsonElement` as its output. It is invoked like any other BeepBeep `Function`, as in the following code example:

```
ParseJson parse = ParseJson.instance;
Object[] out = new Object[1];
parse.evaluate(new Object[]{
    "{\"a\" : 123, \"b\" : false, \"c\" : [4,5,6]}"}, out);
JsonElement j = (JsonElement) out[0];
System.out.println(j);
parse.evaluate(new Object[]{
"{\"a\" : "}, out);
System.out.println(out[0].getClass());
```

✎

We do not illustrate this program, but you can find the symbol used for this function in the glossary at the end of this book. The output of this program is:

```
{"a":123,"b":false,"c":[4,5,6]}
class ca.uqac.lif.json.JsonNull
```

If the parsing fails, such as when the input string is not properly formatted, the function outputs a special `JsonElement` called `JsonNull`, as can be observed

in the second line of output.

`JsonElement` is actually an umbrella class to designate a generic JSON object. In reality, the object returned by the parsing function will belong to one of the descendents of this class, namely:

- `JsonMap` if the parsed string corresponds to an associative map.
- `JsonList` if the parsed string corresponds to a list.
- `JsonString`, `JsonNumber`, or `JsonBoolean` if the string parses to one of the primitive types.

The contents of these objects can also be queried. For example, the following code extracts elements from the object `j` obtained previously, which is actually an instance of `JsonMap`:

```
JsonMap map = (JsonMap) j;
JsonNumber n = (JsonNumber) map.get("a");
System.out.println(n.numberValue());
JsonList l = (JsonList) map.get("c");
System.out.println(l.get(1));
```

✎

The second line of code extracts the value corresponding to key “a” in the map; this value is a `JsonNumber` whose value is printed at the console (`123`). The fourth line of code extracts the value corresponding to key “c” in the map; this is a `JsonList`, in which we `get` the second element (i.e., the element at position 1) and print it at the console (`5`).

JSON objects can be easily queried using these methods. However, suppose we receive a stream of JSON objects, of which we want to extract the value corresponding to some key (say, “a”) and perform further processing on it. This task should be done by an `ApplyFunction` processor –except that `get` is a *method* of a class, not an instance of a BeepBeep `Function`. Thankfully, the JSON palette also provides a second object, called `JPathFunction`. An instance of this function is created by giving it the name of an element to fetch in a JSON object. When it is called on a `JsonElement`, it returns the value corresponding to the given key (or `JsonNull` if the key cannot be found). This function can be passed to an `ApplyFunction` processor, and hence JSON extraction can be applied to a stream of JSON elements. The following code example illustrates this:

```
Object[] out = new Object[1];
```

```
ParseJson.instance.evaluate(new Object[]{
    "{\"a\" : 123, \"b\" : false, \"c\" : [4,5,6]}"}, out);
JsonElement j = (JsonElement) out[0];
JPathFunction f1 = new JPathFunction("a");
f1.evaluate(new Object[]{j}, out);
System.out.println(out[0]);
```

✎

A string is first parsed into a JSON element. A `JPathFunction` is then created, and instructed to fetch the value of a key named “a”. When passed the element `j`, this function returns the `JsonNumber` 123, as expected.

If the field to fetch is nested within another `JsonElement`, it is not necessary to make calls to multiple `JPathFunctions` in succession. As its name implies, the function can accept a *path* expression instead of a single argument. This path is a string that represents a specific traversal inside a JSON element. For example, the expression `c` refers to the path that fetches the value corresponding to key “c”. In the present case, this value is a list; therefore, the path `c[1]` refers to the path that fetches the second value in the list corresponding to the key “c”. This is what is done in the following code example:

```
JPathFunction f2 = new JPathFunction("c[1]");
f2.evaluate(new Object[]{j}, out);
System.out.println(out[0]);
```

✎

By convention, a period is used to designate a value inside a `JsonMap`, while brackets with a number designate a position inside a `JsonList`. Hence, the path `c.e[0].f` would lead to the value “bar” in the JSON document shown at the beginning of this chapter.

XML PARSING

XML parsing and processing works in the same way. As you probably know, XML (the *eXtensible Markup Language*) is another popular notation for storing and exchanging data. An XML document is made of a set of nested “tags” and looks like this:

```
<doc>
  <a>
    <b>1</b>
```

```
    <c>10</c>
  </a>
  <a>
    <b>2</b>
    <c>15</c>
  </a>
  <d>123</d>
  <e>
    <f>foo</f>
    <f>bar</f>
  </e>
</doc>
```

Each tag is enclosed between angle brackets; an *element* is the portion of a document delimited by an opening tag and its corresponding closing tag (these tags have a slash before their name). BeepBeep's XML palette provides a function called `ParseXml` that does the same thing for XML than `ParseJson` does for JSON: it converts a character string into an instance of an object, this time called `XmlElement`, as shown in the following code example:

```
ParseXml parse = ParseXml.instance;
Object[] out = new Object[1];
parse.evaluate(new Object[]{
  "<doc><a>123</a><b>foo</b></doc>"}, out);
XmlElement x = (XmlElement) out[0];
System.out.println(x);
```

✎

The objects returned by the parsing function each have a name, some text (optionally), and a list of children tags (which may be empty). These various fields can be queried as follows:

```
List<XmlElement> ch = (List<XmlElement>) x.getChildren();
XmlElement e = ch.get(1);
System.out.println(e.getName() + ", " + e.getTextElement());
```

✎

The last line of code produces the output `b, foo`.

Parts of an element can also be extracted using a `Function` object similar to `JPathFunction`. It is called `XPathFunction`, and it, too, performs a traversal in a document to retrieve some parts of it. However, since XML documents

are structured differently from JSON, the syntax for writing paths and the actual output of the function are not identical. In its simplest form, a path is a list of tag names separated by slashes. In the XML document shown earlier, evaluating the path `doc/d` would return the `XmlElement` named `d`, which contains the number `1`.

However, there may be multiple elements of the same name; for example, the path `doc/a/e/f` corresponds to two elements: `<f>foo</f>` and `<f>bar</f>`. This is why the evaluation of an XPath expression always returns a *collection*, even when the path corresponds to only one element. The behaviour of the `XPathFunction` is illustrated in the following code example:

```
Object[] out = new Object[1];
ParseXml.instance.evaluate(new Object[]{
    "<doc>\n"
        + "<a><b>1</b><c>10</c></a>\n"
        + "<a><b>2</b><c>15</c></a>\n"
        + "<d>123</d>\n"
        + "</doc>"}, out);
XmlElement x = (XmlElement) out[0];
System.out.println(
    new XPathFunction("doc/d/text()").getValue(x));
System.out.println(
    new XPathFunction("doc/a/b").getValue(x));
System.out.println(
    new XPathFunction("doc/a[b=2]/c").getValue(x));
```

✎

In this excerpt, a few shortcuts are taken: since `XPathFunction` is a descendent of `UnaryFunction`, is has an additional method called `getValuegetValue()` that does away with the usual input/output arrays, and makes for a shorter program. The output of the program is:

```
[123]
[<b>1</b>, <b>2</b>]
[<c>15</c>]
```

The result of the first path is straightforward; however, note the use of `text()` at the end of the path. This is an instruction that extracts the textual content inside the last element. Hence, instead of returning `<d>123</d>` the expression simply returns `123`. It is important to know that `123` is not a `String` object; since the result of an XPath expression is always a collection of `XmlElements`,

the value is encased in a special descendant of this class, called `TextElement`. The textual value that this element contains can be queried using method `toString()`.

The meaning of the second path expression (`doc/a/b`) should be interpreted as: "get all the elements named `<b>` that are inside an element named `<a>`, itself inside an element named `<doc>`". There are indeed two such elements in the input document, but note that the two `<b>`'s do not need to have the same parent `<a>`.

Finally, the third path expression introduces a special notation called a *predicate*, written inside brackets. A predicate is an additional condition on an element, which must be true for this element to be considered in the path. In this example, the condition is that element `a` must have a child called `b` whose textual contents is the value `2`. Therefore, the path expression can be interpreted as: "get all the elements named `<c>` that are inside an element named `<a>` which has a child `<b>` containing value 2, and which is inside an element named `<doc>`. There is indeed a single element satisfying this condition in the document, which is `<c>15</c>`.

Again, the purpose of this section is not to provide an in-depth reference on XML or XPath, which turns out to be a full-fledged query language for XML documents (BeepBeep's palette supports only the basic functionalities of XPath). A last remark must be made on the fact that predicates can contain references to values fetched from a `Context` object. The name of a context key is prefixed by a dollar sign. This is exemplified by the following code:

```
FunctionTree d = new FunctionTree(
    new Bags.ApplyToAll(Numbers.numberCast),
    new XPathFunction("doc/a/b/text()"));
FunctionTree f = new FunctionTree(Numbers.isLessThan,
    new ContextVariable("z"),
    new FunctionTree(Numbers.numberCast,
        new FunctionTree(Bags.anyElement,
            new XPathFunction("doc/a[b=$z]/c/text()"))));
ForAll fa = new ForAll("z", d, f);
```

✎

We first create a function *d* that extracts elements according to the XPath expression `doc/a/b/text()`; this produces a set of `TextElements`. We then call the `Bags` function `ApplyToAll`, which is instructed to cast the elements

of the set into `Numbers` (by applying `NumberCast` on each of them). The end result is that *d* takes as input an XML document, and returns (as numbers) the set of all values found inside a <b> tag.

The second line of code creates another function *f*, which checks that the value of a context variable called *x* is less than another expression on the right-hand side. This expression evaluates the XPath expression `doc/a[b=$z]/c/text()`; note the presence of `$z` in the predicate, which is expected to be replaced by the value of *z* in the current context at the moment the function is evaluated. As before, this expression returns a set of `TextElements`.

Let us assume that the input documents always have a single <c> element inside an <a>. Therefore, the result of the expression will always be a *singleton*: a set with exactly one element. We can take this element out of the set by applying the `Bags` function `AnyElement`, which picks an arbitrary element of a collection. The element is then cast into a number; this is the value that is compared to *x* in the topmost `IsLessThan` function.

Finally, we put functions *d* and *f* inside a `ForAll` quantifier. Graphically, this can be represented in the following figure; the parts of the image that correspond to functions *d* and *f* have been identified.

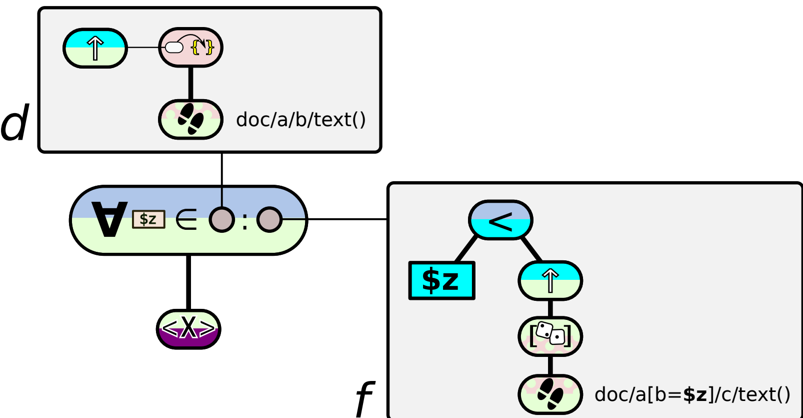


Figure 5.31: Using an XPath expression inside a quantifier.

Given an XML document *x* as input, the quantifier:

- Evaluates function *d* on this document; in this case, it produces a set of numbers corresponding to all the values inside a <b> tag;

- for each number n, it creates a new copy of f, associates the value n to a context key called z, and evaluates $f(x)$;
- it finally computes the logical conjunction of all the returned values.

Informally, object `fa` evaluates the condition: "inside a document, the value of every <b> tag is less than the value of the <c> tag that is located under the same <a> parent". We can try this function on a simple document:

```
FunctionTree d = new FunctionTree(
    new Bags.ApplyToAll(Numbers.numberCast),
    new XPathFunction("doc/a/b/text()"));
FunctionTree f = new FunctionTree(Numbers.isLessThan,
    new ContextVariable("z"),
    new FunctionTree(Numbers.numberCast,
        new FunctionTree(Bags.anyElement,
            new XPathFunction("doc/a[b=$z]/c/text()"))));
ForAll fa = new ForAll("z", d, f);
```

✎

The result produced by `fa` is `true`, as expected. As an exercise, try replacing `2` by `20` in the second <b> tag; you shall see that the quantifier returns the value `false`.

This last example was slightly more involved. However, it gives a foretaste of the wide range of capabilities that become available when one starts mixing objects from multiple palettes. The next chapter shall push the envelope even further on this respect.

Exercises

1. Create a processor chain that takes as input a stream of numbers. Create a scatterplot that shows two lines:

- A first line of (x,y) points where x is a counter that increments by 1 on each new point, and y is the value of the input stream at position x.
- A second line of (x,y) points which is the "smoothed" version of the original. Smoothing can be performed by taking the average of the values at position x-1, x and x+1. As an extra, make it so that the amount of smoothing can be parameterized by a number n, indicating how many events behind and ahead of the current one are included in the average.

2. Modify the second Moore machine example so that the machine outputs the *cumulative* number of times `hasNext()` has been received when `next` is the current input event, and nothing the rest of the time.
3. Create a processor chain whose input events are sets of strings. The chain should return `true` if an event has at least one string of the same length as another one in the previous event, and `false` otherwise.
4. Modify the first example in the *Networking* section, so that the upstream and downstream gateways are in two separate programs. Run the programs of Machine A and Machine B on two different computers. What do you need to change for the communication to succeed?
5. Modify the twin primes example: instead of Machine A pushing numbers to Machine B, make it so that Machine B pulls numbers from Machine A.

6 A Few Use Cases

The previous chapters have introduced a large set of functions and processors, often with very simple code examples illustrating how each of these objects works in isolation. In this chapter, we take a step back and show a "bigger picture". We shall present more complex examples of what can be done when one starts to mix all these objects together. Some of these examples are taken from actual research and development projects where BeepBeep was used, while others have been created especially for this book.

Readers who wish to get more information about these use cases can have a look at some of the research papers on BeepBeep; references are listed at the end of this book.

Stock Ticker

A recurring scenario used in event stream processing to illustrate the performance of various tools is taken from the stock market. One considers a stream of stock quotes, where each event contains attributes such as a stock symbol, the price of the stock at various moments (such as its minimum price and closing price), as well as a timestamp. A typical stream of events of this nature could be the following:

```
1  APPL  1208.4
1  MSFT   800.3
1  GOGL  2001.0
2  APPL  1209.3
2  MSFT   799.7
2  GOGL  2001.1
...
```

Events are structured as tuples, with a fixed set of attributes, each of which taking a scalar value. This simple example can be used to illustrate various queries that typically arise in an event stream processing scenario. A first, simple type of query one can compute over such a trace is called a **snapshot query**, such as the following:

- Get the closing price of MSFT for the first five trading days.

The result of that query is itself a trace of tuples, much in the same way the relational `SELECT` statement on a table returns another table. To illustrate how this query can be executed, we consider the following diagram:

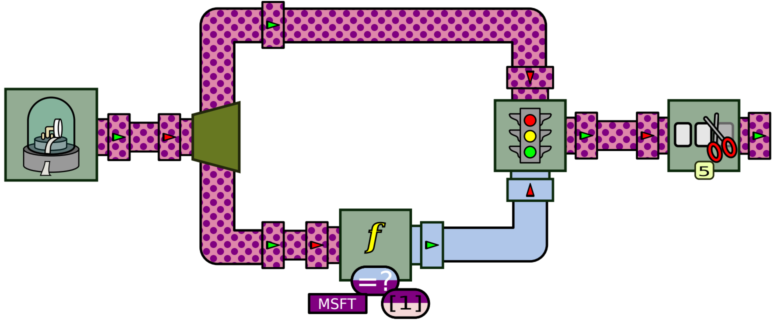


Figure 6.1: Snapshot query.

The first processor box in the figure is a fictitious "ticker source", which, in the present case, generates a random stream similar to the example given above. The events from this source are replicated along two paths. The bottom path is evaluated against the condition that the value in the second column (index 1) is the string "MSFT". The top path is sent into a `Filter` processor, whose control pipe is connected to the stream of Boolean values calculated previously. This results in a stream where only events concerning the MSFT symbol are kept. Finally, the `Prefix` processor retains the first five events from this stream, completing the implementation of the query. This chain of processors corresponds to the following code snippet:

```
TickerFeed feed = new TickerFeed();
Fork fork = new Fork(2);
Connector.connect(feed, fork);
Filter filter = new Filter();
Connector.connect(fork, 0, filter, 0);
ApplyFunction is_msft = new ApplyFunction(
```

```
    new FunctionTree(Equals.instance,
        new Constant("MSFT"),
        new NthElement(1)));
Connector.connect(fork, 1, is_msft, 0);
Connector.connect(is_msft, 0, filter, 1);
Prefix pref = new Prefix(5);
Connector.connect(filter, pref);
```

✎

A refinement of the snapshot query is the **landmark query**, which returns only events that satisfy some criterion, such as:

- Select all the days after the tenth trading day, on which the closing price of MSFT has been greater than $50.

This simple query highlights the fact that, in online processing, outputting a tuple may require waiting until more of the input trace is made available —and that waiting time is not necessarily bounded. In the worst case, MSFT may be the last stock symbol for which the price is known on a given day, and all events of that day must somehow be retained before knowing if they must be output in the result or discarded.

We shall divide this query into two parts. The first part groups all events of the same day into an array, and can be illustrated as follows:

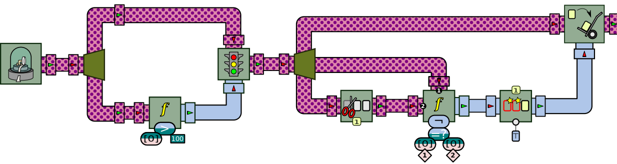


Figure 6.2: Landmark query, part one.

First, events whose timestamp value is lower than 100 are filtered out from the output. Then, a copy of the resulting stream is sent as into the data pipe of a `Pack` processor. Two other copies of the stream are also created; one is delayed by one event, and the timestamp in events of these two copies is then fetched and compared. The end result is a stream that contains the value `false` if an event has the same timestamp as the previous one, and `false` otherwise. A single `false` event is inserted at the beginning of this stream, using an instance of the `Insert` processor. The resulting stream is sent to the control pipe of the `Pack` processor. This corresponds to the following code

snippet:

```
TickerFeed feed = new TickerFeed(10, 20);
Fork fork1 = new Fork(2);
Connector.connect(feed, fork1);
Filter filter = new Filter();
Connector.connect(fork1, 0, filter, 0);
ApplyFunction gt_100 = new ApplyFunction(
    new FunctionTree(Numbers.isGreaterThan,
        new FunctionTree(
            Numbers.numberCast, new NthElement(0)),
        new Constant(10)));
Connector.connect(fork1, 1, gt_100, 0);
Connector.connect(gt_100, 0, filter, 1);
Fork fork2 = new Fork(3);
Connector.connect(filter, fork2);
Lists.Pack pack = new Lists.Pack();
Connector.connect(fork2, 0, pack, 0);
Trim trim = new Trim(1);
Connector.connect(fork2, 1, trim, 0);
ApplyFunction eq = new ApplyFunction(new FunctionTree(Booleans.not,
    new FunctionTree(Equals.instance,
        new FunctionTree(new NthElement(0), StreamVariable.X),
        new FunctionTree(new NthElement(0), StreamVariable.Y))));
Connector.connect(trim, 0, eq, 0);
Connector.connect(fork2, 2, eq, 1);
Insert insert = new Insert(1, false);
Connector.connect(eq, insert);
Connector.connect(insert, 0, pack, 1);
```

✎

The end result is that incoming events are accumulated into a list, until the current event has a different timestamp from the previous one. This triggers the release of the list, and the start of a new one.

The next part of the processor chain checks that, in these created lists, the value of MSFT is at least 50. It can be represented as in Figure 6.3.

This processor chain uses the `RunOn` processor, and evaluates a Boolean condition on each event of the list. This condition checks that, if the stock symbol is MSFT, then its value is greater than 50. This condition returns a distinct Boolean value for each element of the list. The list itself should be considered

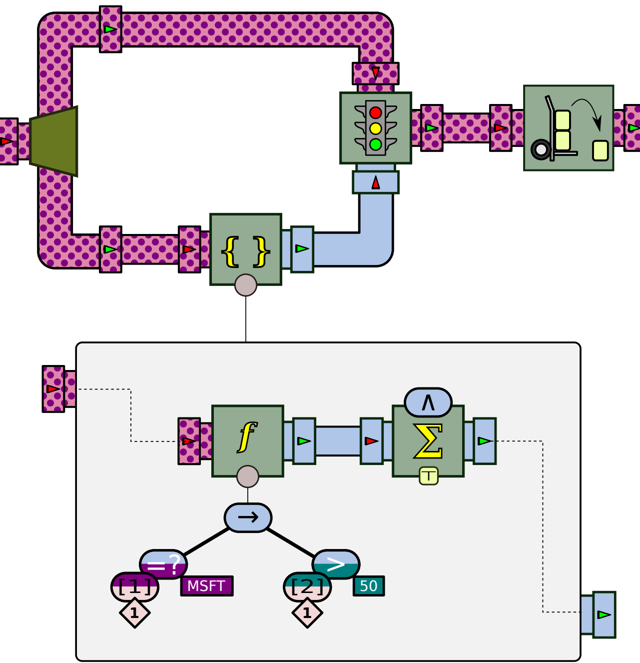


Figure 6.3: Landmark query, part two.

if the condition evaluates to `true` on all its elements. One can evaluate this by sending the output of the `RunOn` processor into a `Cumulate` processor, which is instructed to compute the logical conjunction of the values it receives. This Boolean trace is used as the control stream of a `Filter` processor; only lists where the price for MSFT is higher than 50 will be output. These lists are then sent to an `Unpack` processor, which outputs the elements of each list one by one. In code, this chain can be implemented as in the following snippet:

```
GroupProcessor gp = new GroupProcessor(1, 1);
ApplyFunction ms_50 = new ApplyFunction(
    new FunctionTree(Booleans.implies,
        new FunctionTree(
            Equals.instance,
            new Constant("MSFT"),
            new FunctionTree(
                new NthElement(1), StreamVariable.X)),
        new FunctionTree(Numbers.isGreaterThan,
            new FunctionTree(
```

```
                new NthElement(2), StreamVariable.X),
            new Constant(250))));
gp.addProcessor(ms_50);
gp.associateInput(0, ms_50, 0);
Cumulate c_50 = new Cumulate(
    new CumulativeFunction<Boolean>(Booleans.and));
Connector.connect(ms_50, c_50);
gp.addProcessor(c_50);
gp.associateOutput(0, c_50, 0);
Bags.RunOn ro = new Bags.RunOn(gp);
Fork fork3 = new Fork(2);
Connector.connect(pack, fork3);
Filter f_ms_50 = new Filter();
Connector.connect(fork3, 0, f_ms_50, 0);
Connector.connect(fork3, 1, ro, 0);
Connector.connect(ro, 0, f_ms_50, 1);
Lists.Unpack up = new Lists.Unpack();
Connector.connect(f_ms_50, up);
```

✎

As one can see by examining the output of the program, what comes out of processor `up` is a subset of the original stream containing only events that belong to a day where MSFT closed at $50 or higher. Some queries can also involve aggregate statistics over multiple events:

- On every fifth trading day starting today, calculate the average closing price of MSFT for the five most recent trading days.

As a first step to evaluate this query, we show how to calculate the statistical moment of order n of a set of values, noted $E^n(x)$, in Figure 6.4.

As the diagram shows, the input trace is duplicated into two paths. Along the first (top) path, the sequence of numerical values is sent to the `ApplyFunction` processor computing the n-th power of each value; these values are then sent to a `Cumulate` processor that calculates the sum of these values. Along the second (bottom) path, values are sent to a `TurnInto` processor that transforms them into the constant 1; these values are then summed into another `Cumulative`. The corresponding values are divided by each other, which corresponds to the statistical moment of order n of all numerical values received so far. The average is the case where n=1.

Figure 6.5 shows the chain that computes the average of stock symbol MSFT

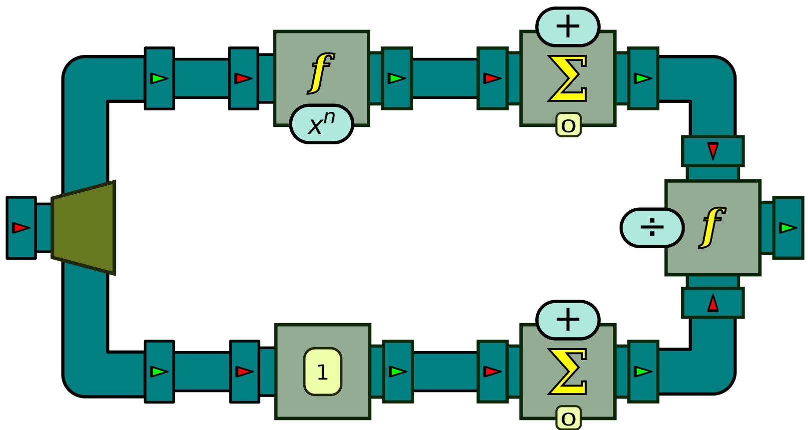


Figure 6.4: A chain of function processors to compute the statistical moment of order n on a trace of numerical events.

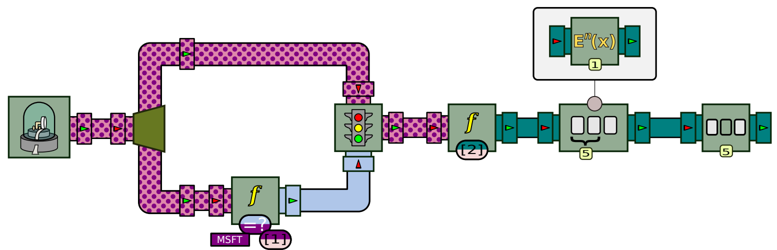


Figure 6.5: Window query.

over a window of 5 events. The first part should now be familiar, and filters events based on their stock symbol. The events that get through this filtering are then converted into a stream of numbers by fetching the value of their closing price. The statistical moment of order 1 is then computed over successive windows of width 5, and one out of every five such windows is then allowed to proceed through the last processor, producing the desired hopping window query. The Java code corresponding to this example is the following:

```
TickerFeed feed = new TickerFeed();
Fork fork = new Fork(2);
Connector.connect(feed, fork);
Filter filter = new Filter();
Connector.connect(fork, 0, filter, 0);
ApplyFunction is_msft = new ApplyFunction(
```

```
    new FunctionTree(Equals.instance,
        new Constant("MSFT"),
        new NthElement(1)));
Connector.connect(fork, 1, is_msft, 0);
Connector.connect(is_msft, 0, filter, 1);
ApplyFunction price = new ApplyFunction(new NthElement(2));
Connector.connect(filter, price);
Window win = new Window(new StatMoment(1), 5);
Connector.connect(price, win);
CountDecimate dec = new CountDecimate(5);
Connector.connect(win, dec);
```

✎

Medical Records Management

We now move to the field of medical record management, where events are messages expressed in a structured format called HL7. An HL7 message is a text string composed of one or more segments, each containing a number of fields separated by the pipe character (|). The possible contents and meaning of each field and each segment is defined in the HL7 specification. The following snippet shows an example of an HL7 message; despite its cryptic syntax, this message has a well-defined, machine-readable structure. However, it slightly deviates from the fixed tuple structure of our first example: although all messages of the same type have the same fixed structure, a single HL7 stream contains events of multiple types.

```
1234567890^DOCLAST^DOCFIRST^M^
^^^^NPI|OBR|1|||80061^LIPID
PROFILE^CPT-4||20070911||||||||||OBX|1|
NM|13457-7^LDL (CALCULATED)^LOINC|
49.000|MG/DL| 0.000 - 100.000|N|||F|
OBX|2|NM|
2093-3^CHOLESTEROL^LOINC|138.000|
MG/DL|100.000 - 200.000|N|||F|OBX|3|
NM|2086-7^HDL^LOINC|24.000|MG/DL|
45.000 - 150.000|L|||F|OBX|4|NM|
2571-8^TRIGLYCERIDES^LOINC|324.000|
```

HL7 messages can be produced from various sources: medical equipment producing test results, patient management software where individual medi-

cal acts and procedures are recorded, drug databases, etc. For a given patient, the merging of all these various sources produces a long sequence of HL7 messages that can be likened to an event stream. The analysis of HL7 event traces produced by health information systems can be used, among other things, to detect significant unexpected changes in data values that could compromise patient safety.

In this context, a general rule, which can apply to any numerical field, identifies whenever a data value starts to deviate from its current trend:

- Notify the user when an observed data field is three standard deviations above or below its mean.

We call such computations trend queries, as they relate a field in the current event to an aggregation function applied on the past values of that field. Trend queries can be made more complex, and correlate values found in multiple events, such as the following:

- Notify the user when two out of three successive data points lie more than two standard deviations from the mean on the same side of the mean line.

Although our example query does not specify it, this aggregation can be computed over a window as defined in our previous use case, such as the past 100 events, or events of the past hour.

Let us show how this query can be computed using chains of function processors. We can reuse the statistical moment processor $E^n(x)$ defined above. Equipped with such processors, the desired property can be evaluated by the graph shown in the next figure (✎).

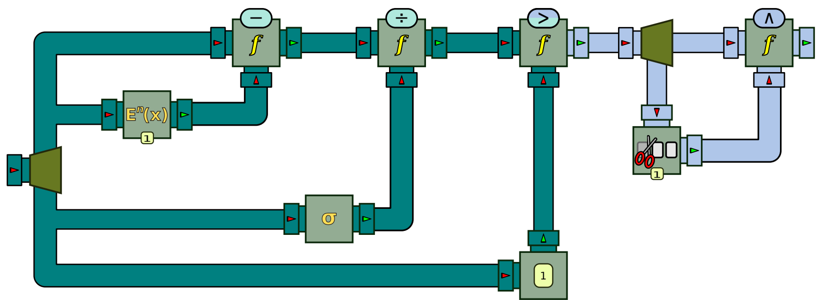


Figure 6.6: The chain of processors for Query 6.

The input trace is divided into four copies. The first copy is subtracted by the statistical moment of order 1 of the second copy, corresponding to the distance of a data point to the mean of all data points that have been read so far. This distance is then divided by the standard deviation (computed form the third copy of the trace). An `ApplyFunction` processor then evaluates whether this value is greater than the constant trace with value 1.

The result is a trace of Boolean values. This trace is itself forked into two copies. One of these copies is sent into a `Trim` processor, which removes the first event of the input trace; both paths are sent to a processor computing their logical conjunction. Hence, an output event will have the value `true` whenever an input value and the next one are both more than two standard deviations from the mean.

Online Auction System

Our next use case moves away from traditional CEP scenarios, and considers a log of events generated by an online auction system [17]. In such a system, when an item is being sold, an auction is created and logged using the start(i, m, p) event, where m is the minimum price the item named i can be sold for and p_is the number of days the auction will last. The passing of days is recorded by a propositional *endOfDay* event; the period of an auction is considered over when there have been p number of *endOfDay* events.

The auction system generates a log of events similar to the following:

```
start(vase,3,15).
bid(vase,15).
start(ring,5,30).
endOfDay.
bid(ring,32).
bid(ring,33).
bid(vase,18).
sell(vase).
```

Although the syntax differs, events of this scenario are similar to the HL7 format: multiple event types (defined by their name) each define a fixed set of attributes.

One could imagine various queries involving the windows and aggregation functions mentioned earlier. However, this scenario introduces special types

of queries of its own. For example:

- Check that every bid of an item is higher than the previous one, and report to the user otherwise.

This query expresses a pattern correlating values in pairs of successive bid events: namely, the price value in any two bid events for the same item *i* must increase monotonically. Some form of slicing, as shown earlier, is obviously involved, as the constraint applies separately for each item; however, the condition to evaluate does not correspond to any of the query types seen so far. A possible workaround would be to add artificial timestamps to each event, and then to perform a join of the stream with itself on *i*: for any pair of bid events, one must then check that an increasing timestamp entails an increasing price. Unfortunately, in addition to being costly to evaluate in practice, stream joins are flatly impossible if the interval between two bid events is unbounded. A much simpler —and more practical— solution would be to simply "freeze" the last *Price* value of each item, and to compare it to the next value. For this reason, queries of that type are called freeze queries.

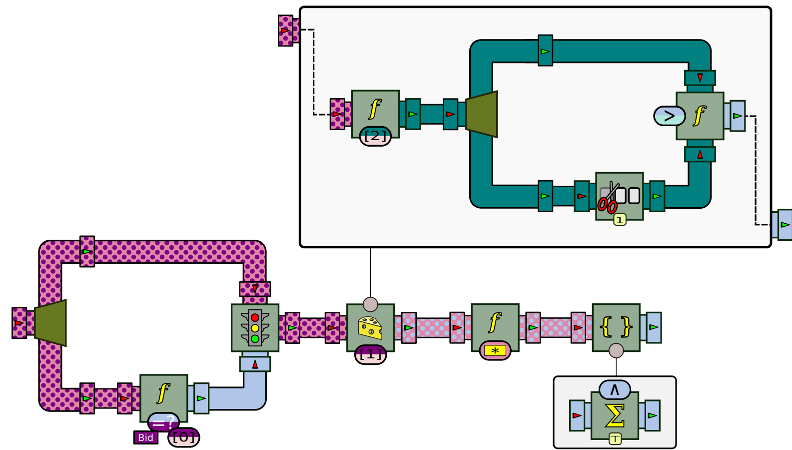


Figure 6.7: Checking that every bid is higher than the previous one.

```
Fork f = new Fork(2);
Connector.connect(split, f);
Filter filter =  new Filter();
Connector.connect(f, 0, filter, 0);
ApplyFunction is_bid = new ApplyFunction(
    new FunctionTree(Equals.instance,
```

```
        new Constant("Bid"),
        new NthElement(0)));
Connector.connect(f, 1, is_bid, 0);
Connector.connect(is_bid, 0, filter, 1);
GroupProcessor bid_amount = new GroupProcessor(1, 1);
{
  ApplyFunction get_amt = new ApplyFunction(new NthElement(2));
  Fork b_f = new Fork(2);
  Connector.connect(get_amt, b_f);
  ApplyFunction gt = new ApplyFunction(Numbers.isLessThan);
  Connector.connect(b_f, 0, gt, 0);
  Trim trim = new Trim(1);
  Connector.connect(b_f, 1, trim, 0);
  Connector.connect(trim, 0, gt, 1);
  bid_amount.associateInput(0, get_amt, 0);
  bid_amount.associateOutput(0, gt, 0);
  bid_amount.addProcessors(get_amt, b_f, gt, trim);
}
Slice slice = new Slice(new NthElement(1), bid_amount);
Connector.connect(filter, slice);
ApplyFunction values = new ApplyFunction(Maps.values);
Connector.connect(slice, values);
Bags.RunOn and = new Bags.RunOn(new Cumulate(
    new CumulativeFunction<Boolean>(Booleans.and)));
Connector.connect(values, and);
```

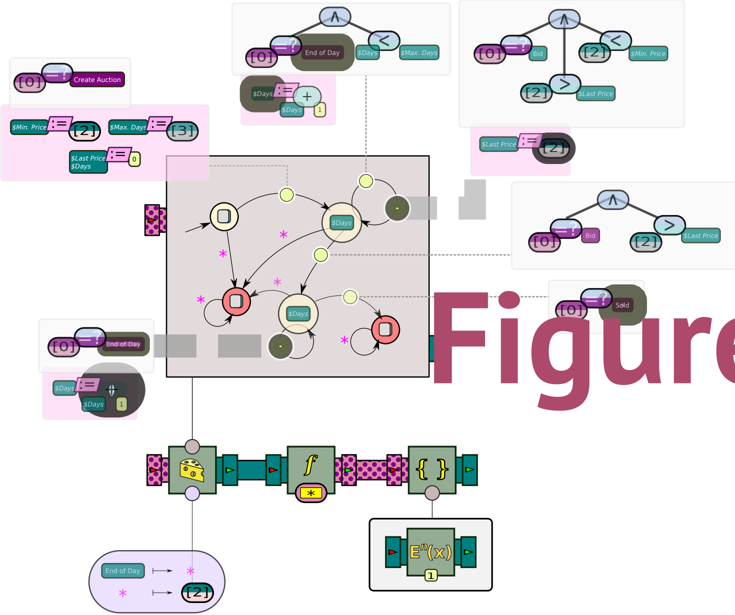
✎

The previous query involved a simple sequential pattern of two successive bid events. However, the auction scenario warrants the expression of more intricate patterns involving multiple events and multiple possible orderings:

- List the items that receive bids outside of the period of their auction.

As one can see, this query refers to the detection of a pattern that takes into account the relative positioning of multiple events in the stream: an alarm should be raised if, for example, a bid for some item *i* is seen before the start event for that same item *i*. Similarly, an occurrence of a bid event for *i* is also invalid if it takes place *n endOfDay* events after its opening, with *n* being the *Duration* attribute of the corresponding start event. We call such query a lifecycle query, as the pattern it describes corresponds to a set of event sequences, akin to what a finite-state machine or a regular expression can express.

Rather than simply checking that the sequencing of events for each item is followed, we will take advantage of BeepBeep's flexibility to compute a non-Boolean query: the average number of days since the start of the auction, for all items whose auction is still open and in a valid state. The processor graph is shown below.



$\rightarrow 0$, $Days \rightarrow 0$. The values of Min. Price and Max. Days are set with the content of the third and fourth element of the tuple, respectively. The remaining transitions take care of updating the minimum price and the number of days elapsed according to the events received.

Each state of the Moore machine is associated with an output value. For three of these states, the value to output is the empty event, meaning that no output should be produced. For the remaining two states, the value to output is the current content of Days, as defined in the processor's context.

According to the semantics of the `Slice` processor, each output event will consist of a set, formed by the last output of every instance of the Moore machine. Thus, this set will contain the number of elapsed days of all items whose auction is currently open (the Moore machine for the other items outputs no number). This set is then passed to a function processor, which computes the average of its values (sum divided by cardinality).

As a bonus, we show how to plot a graph of the evolution of this average over time. We fork the previous output; one branch of this fork goes into a Mutator, which turns the set into the value 1; this stream of `1`s is then sent to a `Cumulate` processor that computes their sum. Both this and the second branch of the fork are fed into a function processor, which creates a named tuple where x is set to the value of the first input, and y is set to the value of the second input. The result is a tuple where x is the number of input events, and y is the average computed earlier. These tuples are then accumulated into a set with the means of another cumulative function processor, this time performing the set addition operation. The end result is a stream of sets of (x,y) pairs, which could then be sent to a `Scatterplot` processor to be plotted with the help of the MTNP palette.

Voyager Telemetry

In this section, we study the data produced by the *Voyager 2* space probe. This automatic probe was launched by NASA in 1977 on a trajectory that allowed it to fly close to four planets of the solar system: Jupiter, Saturn, Uranus and Neptune. On this "Grand Tour" of the solar system, Voyager 2 (along with its twin, Voyager 1) collected scientific data and took pictures that greatly expanded our knowledge of the gas giants and their moons.

As of the time of the writing of this book, both Voyagers are still operational, and currently explore the outer edge of the solar system. The telemetry sent back by these probes, going all the way back to 1977, is publicly available in the form of various text files on a NASA FTP archive. In our example, we shall use a simple, collated dataset that can be downloaded from the following URL:

```
ftp://spdf.gsfc.nasa.gov/pub/data/voyager/voyager2/merged/
```

The files contained in that repository are named `vy2_YYYY.asc`, where `YYYY` corresponds to a year. These files provide averaged hourly readings of various instruments in the spacecraft. One line of such a file looks like this:

```
1977 365 22   1.91    0.6    1.2   ...
```

A file that accompanies the repository describes the meaning of each column. For the purpose of this example, we are only interested in the first four columns, which respectively represent the year, decimal day, hour (0-23) and spacecraft's distance to the Sun expressed in Astronomical Units (AU). From this data, let us see if we can detect the **planetary encounters** of Voyager 2, by looking at how its speed changes over time.

Our long processor chain can be broken into three parts: preprocessing, processing, and visualization.

PREPROCESSING

Preprocessing starts from the raw data, and formats it so that the actual computations are then possible. In a nutshell, the preprocessing step amounts to the following processor chain:

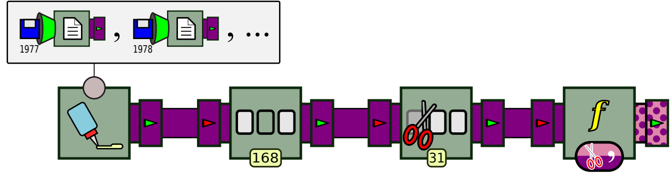


Figure 6.9: Preprocessing the Voyager data.

Since the data is split into multiple CSV files, we shall first create one instance of the `ReadLines` processor for each file, and put these `Sources` into an array. We can then pass this to a processor called `Splice`, which is the first processor box shown in the previous picture. The splice pulls events from the first source

it is given, until that source does not yield any new event. It then starts pulling events from the second one, and so on. This way, the contents of the multiple text files we have can be used as an uninterrupted stream of events. This is why the `Splice` pictogram is a small bottle of glue.

We then perform a drastic reduction of the data stream. The input files have hourly readings, which is a degree of precision that is not necessary for our purpose. We keep only one reading per week, by applying a `CountDecimate` that keeps one event every 168 (there are 168 hours in a week). Moreover, the file corresponding to year 1977 has no meaningful data before week 31 or so (the launch date); we ignore the first 31 events of the resulting stream by using a `Trim`. Finally, as a last preprocessing step, we convert plain text events into arrays by splitting each string on spaces. This is done by applying the `SplitString` function. The Java code of this first preprocessing step looks like this:

```
int start_year = 1977, end_year = 1992;
ReadLines[] readers = new ReadLines[end_year - start_year + 1];
for (int y = start_year; y <= end_year; y++) {
  readers[y - start_year] = new ReadLines(
      PlotSpeed.class.getResourceAsStream(
          "data/vy2_" + y + ".asc"));
}
Splice spl = new Splice(readers);
CountDecimate cd = new CountDecimate(24 * 7);
Connector.connect(spl, cd);
Trim ignore_beginning = new Trim(31);
Connector.connect(cd, ignore_beginning);
ApplyFunction to_array = new ApplyFunction(
    new Strings.SplitString("\\s+"));
Connector.connect(ignore_beginning, to_array);
```

✎

PROCESSING

The next step is to perform computations on this stream of arrays. The goal is to detect rapid variations in the spacecraft's speed, and to visualize these variations in a plot. To this end, the input stream will be forked in three parts, as shown in the following diagram:

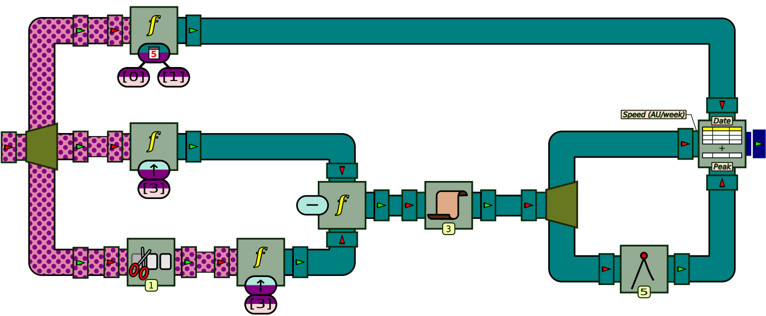


Figure 6.10: Processing the Voyager data.

In the first copy of the stream, we apply a `FunctionTree` which extracts the first element of the input array (a year), the second element of the array (the number of a day in the year), and passes these two values to a custom function called `ToDate`, which turns them into a single number. This number corresponds to the number of days elapsed since January 1st, 1977 (the first day in the input files). Converting the date in such a format will make it easier to plot afterwards. This date is then fed into an `UpdateTableStream` processor, and will provide values for the first column of a three-column table.

In the second copy of the stream, we extract the fourth component of the input array and convert it into a number. This number corresponds to the spacecraft's distance. The third copy of the stream is trimmed from its first event, and the distance to the Sun is also extracted. The two values are then subtracted. The end result is a stream of numbers, representing the difference in distance between two successive events. Since events are spaced by exactly one week, this value makes a crude approximation of the spacecraft's weekly speed.

However, since the weekly distance is very close to the measurement's precision, we "smooth" those values by replacing them by the average of each two successive points. This is the task of the `Smooth` processor, represented in the diagram by a piece of sandpaper.

This stream is again separated in two. The first copy goes directly into the table, and provides the values for its second column. The second copy first goes into a `PeakFinder` processor from the *Signal* palette, before being sent into the table as its third column. The end result is a processor chain that populates a table containing:

- The number of days since 1/1/1977
- The smoothed weekly speed
- The peaks extracted from the weekly speed

In code, this chain of processor looks as follows:

```
Fork fork = new Fork(3);
Connector.connect(to_array, fork);
ApplyFunction format_date = new ApplyFunction(new FunctionTree(
    FormatDate.instance, new FunctionTree(
        new NthElement(0), StreamVariable.X),
    new FunctionTree(new NthElement(1), StreamVariable.X)));
Connector.connect(fork, 0, format_date, INPUT);
ApplyFunction get_au1 = new ApplyFunction(new FunctionTree(
    Numbers.numberCast, new FunctionTree(
        new NthElement(3), StreamVariable.X)));
Connector.connect(fork, 1, get_au1, INPUT);
Trim cd_delay = new Trim(1);
Connector.connect(fork, 2, cd_delay, INPUT);
ApplyFunction get_au2 = new ApplyFunction(new FunctionTree(
    Numbers.numberCast, new FunctionTree(
        new NthElement(3), StreamVariable.X)));
Connector.connect(cd_delay, get_au2);
ApplyFunction distance = new ApplyFunction(new FunctionTree(
    Numbers.maximum, Constant.ZERO, new FunctionTree(
        Numbers.subtraction,
        StreamVariable.X, StreamVariable.Y)));
Connector.connect(get_au2, OUTPUT, distance, TOP);
Connector.connect(get_au1, OUTPUT, distance, BOTTOM);
Smooth smooth = new Smooth(2);
Connector.connect(distance, smooth);
Fork f2 = new Fork(2);
Connector.connect(smooth, f2);
PeakFinderLocalMaximum peak = new PeakFinderLocalMaximum(5);
Connector.connect(f2, BOTTOM, peak, INPUT);
Threshold th = new Threshold(0.0125f);
Connector.connect(peak, th);
Limit li = new Limit(5);
Connector.connect(th, li);
UpdateTableStream table = new UpdateTableStream("Date",
    "Speed (AU/week)", "Peak");
Connector.connect(format_date, OUTPUT, table, 0);
Connector.connect(f2, OUTPUT, table, 1);
```

```
Connector.connect(li, OUTPUT, table, 2);
```

✎

VISUALIZATION

The last step is to display the contents of this table graphically. This can be done using the *Widgets* palette, in the following processor chain:

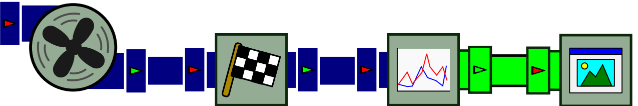


Figure 6.11: Visualizing the Voyager data.

A `Pump` is asked to repeatedly pull on the `UpdateTableStream`; its output is pushed into a `KeepLast` processor; this processor discards all its input events, except when it receives the last one from its upstream source. In this case, this corresponds to a reference the `Table` object once it is fully populated. This table is then passed to a `DrawPlot` processor, and then to a `WidgetSink` in order to be displayed in a `JFrame`. The code producing this chain of processor is as follows:

```
Pump pump = new Pump();
Connector.connect(table, pump);
KeepLast last = new KeepLast(1);
Connector.connect(pump, last);
Scatterplot plot = new Scatterplot();
plot.setCaption(Axis.X, "Days after 1/1/77")
.setCaption(Axis.Y, "AU");
DrawPlot draw = new DrawPlot(plot);
Connector.connect(last, draw);
BitmapJFrame window = new BitmapJFrame();
Connector.connect(draw, window);
window.start();
pump.start();
```

✎

The end result of this program produces a graph, which should look like in the following figure. The blue line shows the craft's average speed, in AU/week, while the green line shows the signal produced by the peak detector. As one

can see, the speed fluctuates relatively smoothly, and the line is interspersed with a few abrupt variations. We can observe that these abrupt changes are picked up by the peak detector, which otherwise outputs a stream of zeros.

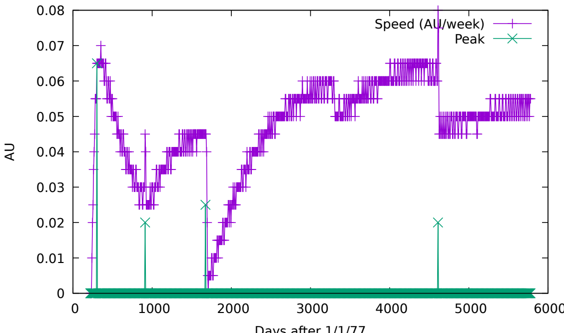


Figure 6.12: A scatterplot created from the Voyager data.

A fun fact about this plot: the last three peaks correspond precisely to the dates of Voyager's flybys of Jupiter, Saturn, and Neptune:

Planet	Date	Days after 1/1/77
Jupiter	July 9, 1979	918
Saturn	August 25, 1981	1,696
Neptune	August 25, 1989	4,618

The flyby of Uranus (January 24, 1986, or Day 3310) does not produce a speed variation large enough to be detected through this method.

Creating this whole chain of processors, from the raw text files to the plot, has required exactly 100 lines of code.

Electric Load Monitoring

The next scenario concerns the concept of *ambient intelligence*, a multidisciplinary approach that consists of enhancing an environment (room, building, car, etc.) with technology (e.g. infrared sensors, pressure mats, etc.), in order to build a system that makes decisions based on real-time information and historical data to benefit the users within this environment. A main challenge of ambient intelligence is activity recognition; it consists in obtaining raw data from sensors, filtering it, and then transforming that data into relevant

information that can be associated with a patient's activities of daily living using Non-Intrusive Appliance Load Monitoring (NIALM). Typically, the parameters considered are the voltage, the electric current and the power (active and reactive). This produces a stream similar to the next figure. An event consists of a timestamp, and numerical readings of each of the aforementioned electrical components.

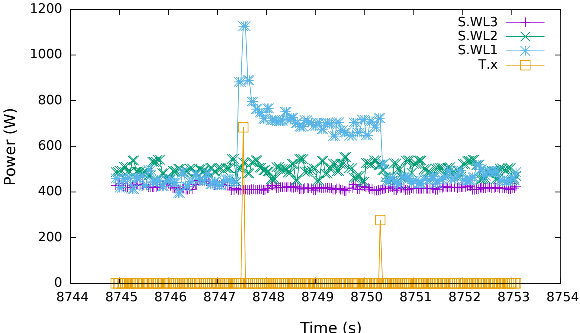


Figure 6.13: The top three lines represent three components of the electrical signal when an electrical appliance is used. In orange, the output of a peak detector taking the electrical signal as its input.

The NIALM approach attempts to associate a device with a load signature extracted from a single power meter installed at the main electrical panel. This signature is made of abrupt variations in one or more components of the electrical signal, whose amplitude can be used to determine which appliance is being turned on or off. An example of query in this context could be:

- Produce a "Toaster On" event when- ever a spike of 1,000±200 W is observed on Phase 1 and the toaster is currently off.

Again, this scenario brings its own peculiarities. Here, events are simple tuples of numerical values, and slicing is applied in order to evaluate each signal component separately; however, the complex, higher-level events to produce depend on the application of a peak detection algorithm over a window of successive time points. Moreover, elements of a lifecycle query can also be found: the current state of each appliance has to be maintained, as the same peak or drop may be interpreted differently depending on whether a device is currently operating or not.

While this scenario certainly is a case of event stream processing in the strictest

sense of the term, it hardly qualifies as a typical CEP scenario, as per the available tools and their associated literature.

The next figure describes the chain of basic event processors that are used to discover the peaks on the electrical signal. The signal from the electrical box is sent to a first processor, which transforms raw readings into name-value tuples, one for each time point. Each tuple contains numerical values for various components of the electrical signal; for example, parameter W1 measures the current active power of Phase 1.

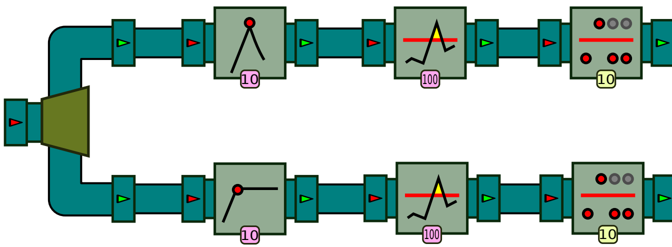


Figure 6.14: The piping of processors for discovering peaks and plateaux on the original electrical signal. Elements in pink indicate parameters that can be adjusted, changing the behaviour of the pipe.

The second processor picks one such parameter from the tuple (W1 in the example), extracts its value, and discards the rest. The output trace from this processor is therefore a sequence of numbers. On the top path, this sequence is then fed to the `PeakFinderLocalMaximum` processor from the *Signal* palette, which detects sudden increases or decreases in a numerical signal. As we have seen in a previous chapter, for each input event, the processor outputs the height of the peak, or the value 0 if this event is not a peak. Since an event needs to be out of the window to determine that it is a peak, the emission of output events is delayed with respect to the consumption of input events.

The next step in the processing takes care of removing some of the noise in the signal. Typical appliances consume at least 100 W and generate a starting peak much higher than that. Therefore, to avoid false positives due to noise, any peak lower than 100 W should be flattened to zero. In order to do so, the output from the peak detector is sent to the `Threshold` processor, set to a threshold value of 100. The resulting trace requires one further cleanup task. Again due to the nature of the electrical signal, two successive peak events may sometimes be reported for the same sudden increase. The last processor takes care of keeping only the first one, using the `Limit` processor.

Given a feed from an electrical signal, this complete chain of processors produces an output trace of numerical events; most of them should be the number 0, and a few others should indicate the occurrence of an abrupt increase or decrease in the values of the input signal, along with the magnitude of that change. Moreover, the position of these events, relative to the original signal, also indicates the exact moment this change was detected. On the lower path of the diagram, the same task is done with the `PlateauFinder` processor to detect plateaux. This corresponds to the following code snippet:

```
Fork f = new Fork(2);
Processor peak_finder = new PeakFinderLocalMaximum(5);
Connector.connect(f, 0, peak_finder, 0);
Threshold peak_th = new Threshold(100);
Connector.connect(peak_finder, peak_th);
Processor peak_damper = new Limit(10);
Connector.connect(peak_th, peak_damper);
Processor plateau_finder = new PlateauFinder()
    .setPlateauRange(5).setRelative(true);
Connector.connect(f, 1, plateau_finder, 0);
Threshold plateau_th = new Threshold(100);
Connector.connect(plateau_finder, plateau_th);
Processor plateau_damper = new Limit(10);
Connector.connect(plateau_th, plateau_damper);
```

✎

The second step is to lift peak and drop events to a yet higher level of abstraction, and to report actual appliances being turned on and off. This is best formalized through the use of a Moore machine, shown in Figure 6.15 (✎).

From the initial state, the event "appliance on" (I) is output only if a peak and a plateau event of the appropriate magnitude are received in immediate succession. At this point, the event "appliance off" (O) is emitted only if a drop of the appropriate magnitude is received. All other input events processed by the machine result in a dummy output event, indicating "No change", being produced. Apart from the actual numerical values, this Moore machine is identical for all appliances. Notice how the abstraction performed in Step 1 simplifies the problem in Step 2 to the definition of a simple, five-state automaton.

The last step is to apply this Moore machine on some electrical signal. To this end, we shall reuse the signal generator used in Chapter 5 to illustrate the

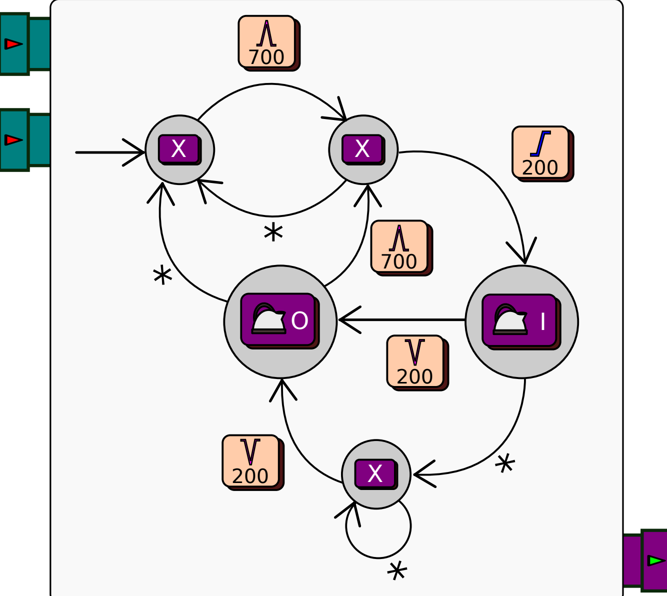


Figure 6.15: The Moore machine for detecting on/off events for a single appliance.

operation of various processors of the *Signal* palette (✎). In this example, we instruct the generator to produce a signal that starts at zero, increases rapidly to the value 1,000, stabilizes at the value 700 and then drop back to zero. This stream is then fed to the signal processing chain described earlier, which produces a "peak" and a "plateau" stream. The two streams then enter the Moore machine, which is instantiated with the signature (peak-plateau-drop) that should be detected. Finally, the time stream of the signal generator is merged with the output of the Moore machine to create tuples of the form (t,s), where t is a timestamp, and s is the detected state of the appliance for that timestamp. This corresponds to the diagram in Figure 6.16.

As one can see, the signal processing chain has been encapsulated into a single box, with numbers at its edges representing the parameters given to the processors encased into it. Similarly, the Moore machine is also represented as a single box, with the numbers 1,000, 700 and -700 representing the values used in the conditions on state transitions. In code, the remaining steps can be written like this:

```
ApplianceMooreMachine amm =
```

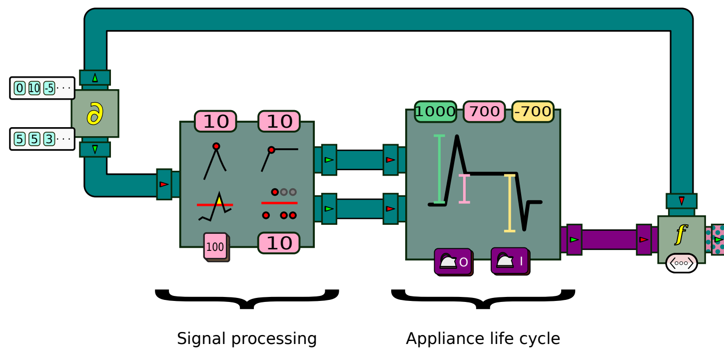


Figure 6.16: The complete detection process.

```
    new ApplianceMooreMachine(1000, 700, -700, 150);
Connector.connect(peak_damper, 0, amm, 0);
Connector.connect(plateau_damper, 0, amm, 1);
GenerateSignalNoise signal = new GenerateSignalNoise(10f,
    new Object[] {0, 333, -150, 0, -175, 0},
    new Object[] {70, 3, 2, 100, 4, 60});
Connector.connect(signal, 1, f, 0);
ApplyFunction to_list = new ApplyFunction(
    new Bags.ToList(Number.class, Number.class));
Connector.connect(signal, 0, to_list, 0);
Connector.connect(amm, 0, to_list, 1);
```

✎

Running this program will print at the console:

```
[1.0, No change]
[2.0, No change]
[3.0, No change]
...
```

With the default values given to the program, the Moore machine should output the string “Appliance ON” around timestamp 75, and “Appliance OFF” around timestamp 180. Indeed, this corresponds to the approximate moments, in the input signal, of the simulated spikes and drops.

Video Game

Our last use case considers event streams produced by the execution of a piece of software. Runtime verification is the process of observing a sequence of events generated by a running system and comparing it to some formal specification for potential violations. It was shown how the use of a runtime monitor can speed up the testing phase of a system, such as a video game under development, by automating the detection of bugs when the game is being played.

We take as an example the case of a game called *Pingus*, a clone of Psygnosis' Lemmings game series. The game is divided into levels populated with various kinds of obstacles, walls, and gaps. Between 10 and 100 autonomous, penguin-like characters (the Pingus) progressively enter the level from a trapdoor and start walking across the area. The player can give special abilities to certain Pingus, allowing them to modify the landscape to create a walkable path to the goal. For example, some Pingus can become Bashers and dig into the ground; others can become Builders and construct a staircase to reach over a gap. The following figure shows a screenshot of the game.

Figure 6.17: A screenshot of the Pingus video game in action.

When running, the game updates the playing field about 150 times per second; each cycle of the game's main loop produces an XML snapshot of its state similar to the one shown below:

```
<event>
  <characters>
    <character>
      <id>0</id>
      <status>Walker</status>
      <position>
        <x>100</x>
        <y>81</y>
      </position>
      <velocity>
        <x>-0.5</x>
        <y>0.3</y>
      </velocity>
    </character>
    ...
  </characters>
</event>
```

Hence, analyzing the execution of the game can be assimilated to processing the stream of individual XML events it generates. The abnormal execution of the game can be expressed as a set of event stream queries, looking for a pattern corresponding to bugs in the game. An example of an incorrect execution pattern could be:

- Make sure that a walking Pingu that encounters a Blocker turns around and starts walking in the other direction.

This query is special in at least two respects. First, the Pingus use case introduces a new type of event unseen in previous examples. Indeed, the XML events produced by the game are not fixed tuples of name-value pairs, but rather contain nested substructures. Hence, in each event, the `<character>` element is repeated for as many Pingus as there are on the playing field; each such element contains the data (position, velocity, skills) specific to one character. It does not make sense, in this context, to talk about "the" ID inside an event, as it contains multiple such IDs. The contents of XML documents must therefore be accessed using a more sophisticated querying mechanism, such as XPath expressions. Moreover, events are unusually large: a single event can contain as much as ten kilobytes of XML data.

Second, in order to detect this pattern of events, one must correlate the x-y position of two distinct Pingus (a Walker and a Blocker), and then make sure that the distance between the two increases over the next couple of

events (indicating a turnaround). An encounter occurs whenever the (x,y) coordinates of the Walker come within 6 pixels horizontally, and 10 pixels vertically, of some Blocker. When this happens, the Walker may continue walking towards the Blocker for a few more events, but eventually turns around and starts walking away.

The following diagram shows the processor graph that verifies this. Here, blue pipes carry XML events, turquoise pipes carry events that are scalar numbers, and grey pipes contain Boolean events.

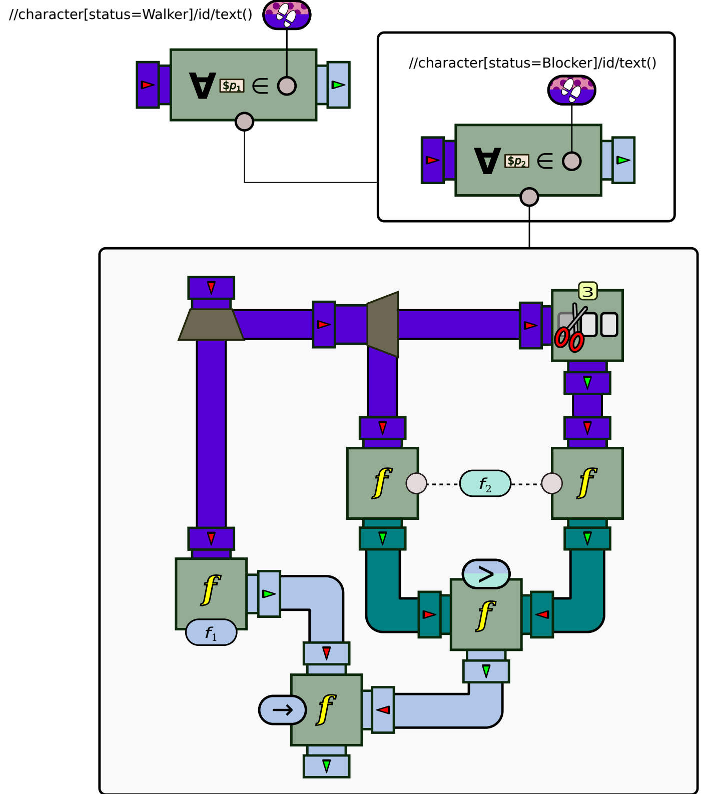


Figure 6.18: The processor chain for processing a stream of events for the Pingus video game.

The XML trace is first sent into a universal quantifier. The domain function, represented by the oval at the top, is the evaluation of the XPath expression `//character[status=Walker]/id/text()` on the current event; this fetches the value of attribute id of all characters whose status is `Walker`. For every such

value c, a new instance of the underlying processor will be created, and the context of this processor will be augmented with the association $p_1 \rightarrow c$. The underlying processor, in this case, is yet another quantifier. This one fetches the ID of every `Blocker`, and for each such value c_0, creates one instance of the underlying processor and adds to its context the association $p_2 \rightarrow c_0$.

The underlying processor is the graph enclosed in a large box at the bottom. It creates two copies of the input trace. The first goes to the input of a function processor evaluating function f_1 on each event. This function evaluates the conjunction of the two conditions $|x_1 - x_2| < 6$ and $|y_1 - y_2| < 10$, where x_i and y_i are the coordinates of the Pingu with ID p_i. Function f_1 is the `FunctionTree` described in the following diagram (✎):

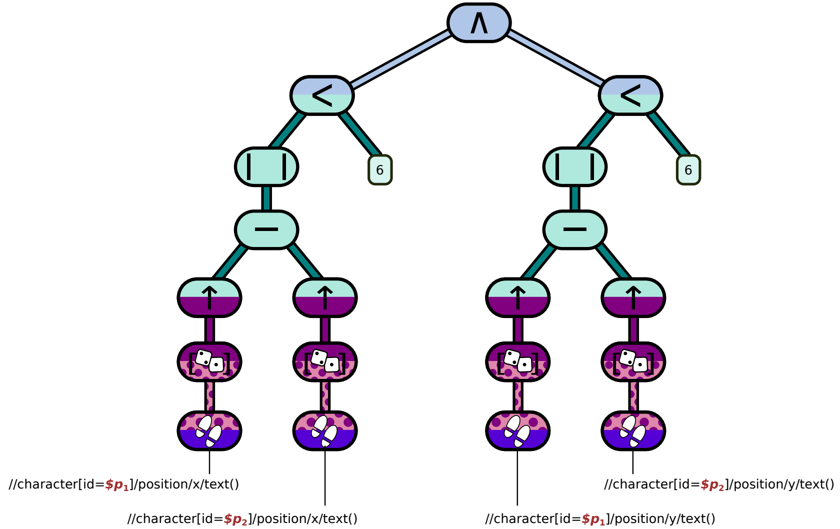


Figure 6.19: Function f_1 expresses a condition on the x-y distance between two Pingus with IDs $p1 and $p2.

Its left branch fetches the x position of characters with ID p_1 and p_2, and checks whether their absolute difference is greater than 6. Its right branch (not shown) does a similar comparison with the y position of both characters. Note in this case how the XPath expression to evaluate refers to elements of the processor's context (p_1 and p_2). The resulting function returns a Boolean value, which is true whenever character p_1 collides with p_2.

The second copy of the input trace is duplicated one more time. The first is sent to a function processor evaluating f_2, which computes the horizontal

distance between p_1 and p_2, as is shown in the following diagram (✎):

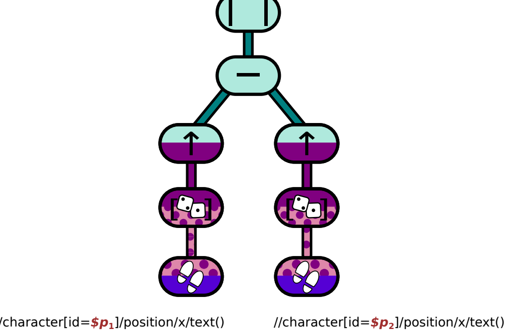


Figure 6.20: Function f_2 computes the horizontal distance between two Pingus with IDs $p1 and $p2.

The second is sent to the `Trim` processor, which is instructed to remove the first three events it receives and lets the others through. The resulting trace is also sent into a function processor evaluating f_2. Finally, the two traces are sent as the input of a function processor evaluating the condition >. Therefore, this processor checks whether the horizontal distance between p_1 and p_2 in the current event is smaller than the same distance three events later. If this is true, then p_1 moved away from p_2 during that interval.

The last step is to evaluate the overall expression. The "collides" Boolean trace is combined with the "moves away" Boolean trace in the `Implies` processor. For a given event e, the output of this processor will be `true` when, if p_1 and p_2 collide in e, then p_1 will have moved away from p_2 three events later.

Furthermore, various kinds of analyses can also be conducted on the execution of the game. For example, one may be interested in watching the real time number of Pingus possessing a particular skill, leading to a query such as:

- Determine the real time proportion of all active Pingus that are Blockers.

Such a query involves, for each event, the counting of all Pingus with a given skill with respect to the total number of Pingus contained in the event. This is a much easier query than the previous one; it can be implemented as in the diagram shown in Figure 6.21.

First, an XPath function retrieves the list of all distinct values of the `<status>` element inside each event. This list of values is then sent into a `Window` pro-

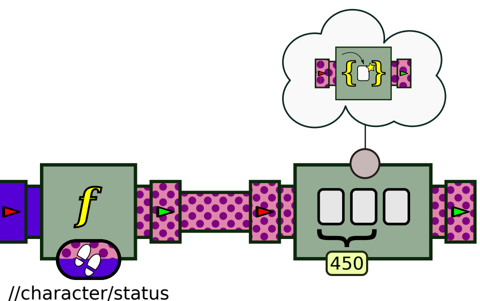


Figure 6.21: The processor chain to compute the proportion of Pingus of each skill.

cessor, which accumulates them into a `Multiset`. This object behaves in a way similar to a set, except that the multiplicity of the elements inside the set is preserved. Once a window of 450 events is complete, this multiset is output by the processor. In code, this corresponds to the following chain:

```
ApplyFunction skill = new ApplyFunction(
    new XPathFunction("message/characters/character/status/text()"));
Connector.connect(x_reader, skill);
GroupProcessor gp = new GroupProcessor(1, 1);
{
  Lists.Unpack unpack = new Lists.Unpack();
  Multiset.PutInto pi = new Multiset.PutInto();
  Connector.connect(unpack, pi);
  gp.addProcessors(unpack, pi);
  gp.associateInput(0, unpack, 0);
  gp.associateOutput(0, pi, 0);
}
Window win = new Window(gp, 2);
Connector.connect(skill, win);
```

✎

The expected output of this program is something that looks like the following:

```
{WALKER=188, BLOCKER=12}
{WALKER=187, BASHER=1, BLOCKER=12}
...
```

As one can see, each event corresponds to a multiset giving the number of Pingus of each skill, in a window of 450 successive events.

Exercises

1. Implement the queries that are mentioned in this chapter but not shown in detail.
2. In the Voyager example, modify the processor chain so that:

- a. The files are read directly from the FTP site.
- b. The plot is written to a file instead of being displayed in a window.
- c. The plot updates after every year processed, instead of at the end.

3. Modify the NIALM example to detect appliances based on a signature that involves more than one signal at a time.
4. In the Pingus example, divide the playing field into square cells of a given number of pixels, and count the Pingus that lie in each cell at any given moment, producing a form of "heat map".

7 Extending BeepBeep

BeepBeep was initially designed to be easily extensible. As was discussed earlier, it consists of only a small core of built-in processors and functions. The rest of its functionalities are implemented through custom processors and grammar extensions, grouped in packages called *palettes*.

However, it is quite possible that none of the processors or functions in existing palettes are appropriate for a particular problem. Fortunately, BeepBeep provides easy means to create your own objects, by simply extending some of the classes provided by the core library. Through multiple examples contained in this chapter, we shall see how custom functions and processors can be created, often in just a few lines of code.

Creating Custom Functions

Let us start with the simple case of functions. A custom function is any object that inherits from the base class `Function`. There are two main ways to create new function classes:

- By extending `FunctionTree` and composing existing `Function` objects
- By extending `Function` or one of its more specific descendents, such as `UnaryFunction` or `BinaryFunction`. In such a case, the function can made of arbitrary Java code.

AS A FUNCTION TREE

A first way of creating a function is to create a new class that extends `FunctionTree`. Remember the function tree that was created in Chapter 3, which

computed the function $f(x,y,z) = (x+y) \times z$. We used to build this function tree as follows:

```
FunctionTree tree = new FunctionTree(Numbers.multiplication,
                new FunctionTree(Numbers.addition,
                        StreamVariable.X, StreamVariable.Y),
                StreamVariable.Z);
```

However, if this function needs to be reused in various programs, the previous instruction has to be copy-pasted multiple times –which creates all the problems associated with copy-pasting. A better practice would be to create a `CustomFunctionTree` that encapsulates the creation of the function inside its constructor. This can be done by creating a new class that extends `FunctionTree`, like this:

```
public class CustomFunctionTree extends FunctionTree
{
    public CustomFunctionTree()
    {
        super(Numbers.multiplication,
                new FunctionTree(Numbers.addition,
                        StreamVariable.X, StreamVariable.Y),
                StreamVariable.Z);
    }
}
```

✎

Note how the first `new FunctionTree` in the original code has been replaced by a call to `super`, the constructor of the parent `FunctionTree` class. From then on, `CustomFunctionTree` can be used anywhere like the original function tree, for example like this:

```
Function f = new CustomFunctionTree();
ApplyFunction af = new ApplyFunction(f);
...
```

An interesting advantage is that, if one wishes to change the actual function that is computed, a modification needs to be made in a single location.

AS A NEW OBJECT

The previous technique only works for custom functions that can actually be expressed in terms of existing functions. When this is not possible, users can create a new `Function` class from scratch, composed of arbitrary Java code. It turns out that this is not very hard.

The most generic way of doing so is to directly extend the abstract class `Function`, and to implement all the mandatory methods. There are six of them:

- The `evaluate` method is responsible for doing the actual computation; it receives an array of input arguments, and writes to an array of output arguments.
- The `getInputArity` and `getOutputArity` methods report the function's input and output arity, respectively. They must each return a single integer number.
- The `getInputTypesFor` method is used to specify the type of the function's input arguments. The `getOutputTypeFor` method does the same thing for the function's output values.
- The `duplicate` method must return a new instance (a "clone") of the function.

As a simple example, let us write a new `Function` that multiplies a number by two. We start by creating an empty class that extends `Function`:

```
public class CustomDouble extends Function
{
}
```

A few methods are easy to implement. The case of `getInputArity` and `getOutputArity` can be solved quickly: here, the function is expected to receive one argument, and to produce one output value; hence both methods should return 1. The `duplicate` method is also straightforward: we simply need to return a new instance of `CustomDouble`. This yields the following code:

```
@Override
public int getInputArity()
{
    return 1;
}
@Override
public int getOutputArity()
```

```
{
    return 1;
}
@Override
public Function duplicate(boolean with_state)
{
    return new CustomDouble();
}
```

✎

The next method to implement is `evaluate`, which receives an `inputs` array and an `outputs` array. Since the function reports an input arity of 1, `inputs` should contain a single element; moreover, this element should be an instance of `Number`. Similarly, we expect `outputs` to be an array of size 1. The method produces its return value by writing to the `outputs` array. The code for `evaluate` could therefore look like this:

```
public void evaluate(Object[] inputs, Object[] outputs)
{
    Number n = (Number) inputs[0];
    outputs[0] = n.floatValue() * 2;
}
```

✎

The `getInputTypesFor` method allows other objects to query the function about the type of its arguments. It receives a set *s* of classes and an index *i* as arguments; its task is to add to *s* the `Class` object corresponding to the expected type of the *i*-th argument of the function (as usual, indexes start at 0). This results in the following code:

```
public void getInputTypesFor(Set<Class<?>> s, int i)
{
    if (i == 0)
        s.add(Number.class);
}
```

✎

The method checks if *i* is 0; if so, it adds the class `Number` into *s*, otherwise it adds nothing. This is because `CustomDouble` has only one argument; it does not make sense to provide type information for indexes higher than 0. It is important to note that this method can add more than one class to the

set. For example, a function could accept either sets or lists as its arguments; in such a case, method `getInputTypesFor` would add both `List.class` and `Set.class` to the set.

The principle for `getOutputTypeFor` is similar; the slight difference is that the method must *return* a `Class` object:

```
public Class<?> getOutputTypeFor(int i)
{
    if (i == 0)
        return Number.class;
    return null;
}
```

✎

Again, a type is only returned if i=0.

The purpose of the input/output arity/type methods is to declare what is called the function's *signature*. The `Connector` object we use to create processor chains calls these methods to make sure that functions and processors are piped correctly. It is therefore important to properly declare the arity and types for each custom object we create, in order to avoid exceptions being thrown when calling `connect()` with these objects.

This code produces a complete new `Function` object that can be used like any other. For example:

```
Function f = new CustomDouble();
ApplyFunction af = new ApplyFunction(f);
```

As you might expect, a `Function` may have more than one input or output argument; these arguments do not need to be of the same type. To illustrate this, let us create a new function `CutString` that takes two arguments: a string s and a number n. Its purpose is to cut s after n characters and return the result. A possible implementation would be:

```
public class CutString extends Function
{
    public void evaluate(Object[] inputs, Object[] outputs) {
        outputs[0] = ((String) inputs[0]).substring(0,
            (Integer) inputs[1]);
    }

    public int getInputArity() {
```

```
        return 2;
    }

    public int getOutputArity() {
        return 1;
    }

    public Function duplicate(boolean with_state) {
        return new CutString();
    }

    public void getInputTypesFor(Set<Class<?>> s, int i) {
        if (i == 0)
            s.add(String.class);
        if (i == 1)
            s.add(Number.class);
    }

    public Class<?> getOutputTypeFor(int i) {
        if (i == 0)
            return String.class;
        return null;
    }
}
```

✎

UNARY AND BINARY FUNCTIONS

Extending `Function` directly results in lots of "boilerplate" code. If the intended function is 1:1 or 2:1 (that is, it has an input arity of 1 or 2, and an output arity of 1), a shorter way to create a new `Function` object is to create a new class that extends either `UnaryFunction` or `BinaryFunction`. These classes take care of most of the tasks associated to functions, and require the user to simply implement a method called `getValue()`, responsible for computing the output, given some input(s). In this method, the user can write arbitrary Java code.

As an example, let us rewrite the `CustomDouble` function; it is a 1:1 function, which means that it can extend the `UnaryFunction` class. From then on, this new object only requires five lines of code:

```java
public class UnaryDouble extends UnaryFunction<Number,Number>
{
    public UnaryDouble()
    {
        super(Number.class, Number.class);
    }

    @Override
    public Number getValue(Number x)
    {
        return x.floatValue() * 2;
    }
}
```

✎

As you can see, the input and output type for the function must also be declared; here, the function accepts a `Number` and returns a `Number`. These types must also be present in the function's constructor: the superclass constructor must be called, and be given a `Class` instance of each input and output argument.

The `getValue()` method is the one in which the output of the function is computed from the input. Since the function is unary and discloses its single input argument as a number, the method has a single `Number` argument. Similarly, since the function declares its output to also be a number, the return type of this method is `Number`.

Function `CutString` could also be simplified by defining it as a descendent of `BinaryFunction`:

```java
public class BinaryCutString extends
  BinaryFunction<String,Number,String>
{
    public BinaryCutString()
    {
        super(String.class, Number.class, String.class);
    }

    public String getValue(String s, Number n)
    {
        return s.substring(0, n.intValue());
    }
```

```
}
```

✎

This time, the class has three type arguments: the first two represent the type of the first and second argument, and the last represents the type of the return value. Otherwise, the `getValue` method works according to similar principles as `UnaryFunction`.

PARTIAL EVALUATION

In Chapter 4, we saw that functions can also be partially evaluated. As an example, let us create a function that calculates the area of a triangle based on the length of its three sides, by using Heron's formula: if A is the area of the triangle, and a, b, and c are the lengths of its sides, then $A^2 = s(s\text{-}a)(s\text{-}b)(s\text{-}c)$, where s is the *semiperimeter*, or half of the triangle's perimeter. Writing method `evaluate` for this function is relatively straightforward:

```
public void evaluate(Object[] inputs, Object[] outputs)
{
  float a = ((Number) inputs[0]).floatValue();
  float b = ((Number) inputs[1]).floatValue();
  float c = ((Number) inputs[2]).floatValue();
  float s = (a + b + c) / 2f;
  outputs[0] = Math.sqrt(s * (s-a) * (s-b) * (s-c));
}
```

✎

However, a few shortcuts can be made when evaluating this function. For example, as soon as one of the sides is 0, the shape cannot be a triangle, and we can set the area to 0. To implement this functionality, a `Function` object must override a method called `evaluatePartial`, as follows:

```
public boolean evaluatePartial(Object[] inputs,
    Object[] outputs, Context c)
{
  if (inputs[0] != null && ((Number) inputs[0]).floatValue() == 0)
  {
    outputs[0] = 0;
    return true;
  }
  if (inputs[1] != null && ((Number) inputs[1]).floatValue() == 0)
```

```
  {
    outputs[0] = 0;
    return true;
  }
  if (inputs[2] != null && ((Number) inputs[2]).floatValue() == 0)
  {
    outputs[0] = 0;
    return true;
  }
  if (inputs[0] != null && inputs[1] != null && inputs[2] != null)
  {
    evaluate(inputs, outputs);
    return true;
  }
  outputs[0] = null;
  return false;
}
```

✎

The signature of this method contains an array of input arguments, an array of output values, and a `Context` object (which may be null). Since this method is called during partial evaluation, any of the elements of the `inputs` array may be null. Therefore, the method checks, for each of the three elements of the array, whether it is non-null, and if so, whether it is equal to zero. If this is the case, value 0 is put into the `outputs` array, and the method returns `true`. This is meant to indicate that the function was successfully evaluated and produced an output value.

If none of the elements is equal to zero, the method then checks if all the elements are non-null; if so, the method calls `evaluate` to compute its output value using the formula. Finally, when none of these conditions apply, the method returns `false`, indicating that no output value could be computed.

Let us now try partial evaluation using various combinations of input arguments, as in the following program:

```
TriangleArea ta = new TriangleArea();
Object[] out = new Object[1];
boolean b;
b = ta.evaluatePartial(new Object[] {3, 4, 5}, out, null);
System.out.println("b: " + b + ", " + out[0]);
b = ta.evaluatePartial(new Object[] {3, null, 5}, out, null);
```

```
System.out.println("b: " + b + ", " + out[0]);
b = ta.evaluatePartial(new Object[] {3, null, 0}, out, null);
System.out.println("b: " + b + ", " + out[0]);
```

✎

The first case corresponds to regular evaluation; the method returns `true` and puts the area 6 in the `out` array. The second case corresponds to partial evaluation, but where no output value can be computed; consequently, the method returns `false`. Finally, the third case also corresponds to partial evaluation, but where an output value can be produced. Therefore, the output of this program is:

```
b: true, 6.0
b: false, null
b: true, 0
```

Create your Own Processor

As with functions, BeepBeep allows you to create new `Processor` objects, which can then be composed with existing processors. Again, there are multiple ways of creating a new processor:

- As a descendent of `GroupProcessor`, by combining existing processors
- As a descendent of `Processor`, using arbitrary Java code

AS A GROUPPROCESSOR

A first way to create a new processor is to define a class that extends `GroupProcessor`, and to put the instructions building the desired chain of processors into that class' constructor.

Suppose that a user wants to create a processor that counts events. A simple way to do it is to create a `GroupProcessor` as this one:

```
GroupProcessor g = new GroupProcessor(1, 1);
{
    TurnInto one = new TurnInto(1);
    Cumulate sum = new Cumulate(
      new CumulativeFunction<Number>(Numbers.addition));
    Connector.connect(one, sum);
```

```
    g.associateInput(0, one, 0);
    g.associateOutput(0, sum, 0);
    g.addProcessors(one, sum);
}
```

However, if the user wants to use this processor at multiple locations, he will again have to copy-paste this code everywhere a new instance of the counter is needed. A better way is to create a new class that extends `GroupProcessor`:

```
public class CounterGroup extends GroupProcessor
{
    public CounterGroup()
    {
        super(1, 1);
        TurnInto one = new TurnInto(1);
        Cumulate sum = new Cumulate(
            new CumulativeFunction<Number>(Numbers.addition));
        Connector.connect(one, sum);
        associateInput(0, one, 0);
        associateOutput(0, sum, 0);
        addProcessors(one, sum);
    }
}
```

✎

From then on, it is possible to write `new CounterGroup()` to get a fresh instance of this processor.

AS A DESCENDENT OF PROCESSOR

Using a group works only if your custom processor can be expressed by piping other existing processors. If this is not the case, you have to resort to extending one of BeepBeep's `Processor` descendents. The most generic way to do so is to extend the `Processor` class directly. This class defines many functionalities that the user does not need to implement:

- Methods `getInputArity` and `getOutputArity` declare the input and output arity of the processor.
- Based on these arities, the `Processor` class takes care of creating the appropriate number of input and output queues for storing events.

- Method `setPullableInput` associates one of the processor's input pipes to the `Pullable` object of an upstream processor.
- Method `setPushableOutput` associates one of the processor's output pipes to the `Pushable` object of a downstream processor.
- Methods `getContext` and `setContext` handle the interaction with the processor's internal `Context` object.
- Finally, the `Processor` class also handles the unique ID given to each instance, which can be queried with `getId`.

These methods correspond to the very basic functionalities of a BeepBeep processor. As the reader may observe, almost none of these methods need to be called by an end-user creating processor chains (as a matter of fact, none of the code examples we have seen so far use these methods, except for `getId`). They are mostly used by the `Connector` utility class, which, as we have seen, is responsible for piping processor objects together. Many of these methods are declared `final`, which means that their behaviour cannot be changed by descendents of this class. However, since `Processor` itself is abstract, a number of important methods are left to the user to be implemented:

- Method `duplicate` must create a copy of the current processor object.
- Method `getPushableInput` must provide an instance of an object implementing the `Pushable` interface to feed input values when the processor is used in push mode.
- Method `getPullableOutput` must provide an instance of an object implementing the `Pullable` interface to fetch output values when the processor is used in pull mode.

All the event handling functionalities must, of course, be implemented by the user. Typically, this means that the `Pullable` and `Pushable` objects keep a reference to the underlying processor they are associated with; calls to `pull` or `push` trigger some computation inside the processor and manipulate events in the input and output queues. For synchronous processing (which corresponds to almost every processor found in this book), this task is tedious, especially for processors with an input arity greater than 1. For example, a call to `push` may not trigger the computation of an output event if a complete input front cannot be consumed; in push mode, one must also carefully implement the subtle behaviour of the `pull` and `pullSoft` methods, and so on. We do not recommend users to extend this class directly, except perhaps in very specific situations.

Fortunately, BeepBeep provides a descendent of `Processor` that takes care

of even more functionalities for the user; this class is called `SynchronousProcessor`. This class defines its own `Pushable` and `Pullable` objects, and therefore, already implements the `getPushableInput` and `getPullableInput` methods. All the user has left to do is to:

- Decide the input and output arity of the processor; this is done by passing these two numbers to `SynchronousProcessor`'s constructor, typically in a call to `super()` in the new class's constructor.
- Write the actual computation that should occur when a *complete* input front becomes available, i.e. what output event(s) to produce (if any), given an input event; this is done by implementing a method called `compute`.
- Override the method `duplicate` to produce a copy of the processor. (If the processor is stateless, it is recommended to simply return `this`.)
- Optionally, override the methods `getInputTypesFor` and `getOutputTypeFor` (to declare the input and output type for each of the processor's pipes).

Using `SynchronousProcessor`, the minimal working example for a custom processor is made of half-a-dozen lines of code:

```
public class MyProcessor extends SynchronousProcessor
{
  public MyProcessor()
  {
    super(0, 0);
  }

  @Override
  public boolean compute(Object[] inputs, Queue<Object[]> outputs)
  {
    return true;
  }

  @Override
  public Processor duplicate(boolean with_state)
  {
    return this;
  }
}
```

✎

This results in a processor that accepts no inputs, and produces no output. To make things more interesting, we will study a couple of examples.

A SIMPLE 1:1 PROCESSOR

As a first example, let us write a processor that receives character strings as its input event, and computes the length of each string (we know that BeepBeep's `Size` function already does this, but let us ignore it for the purpose of this example). The input arity of this processor is therefore 1 (it receives one string at a time), and its output arity is 1 (it outputs a number). Specifying the input and output arity is done through the call to `super()` in the processor's constructor: the first argument is the input arity, and the second argument is the output arity.

The actual functionality of the processor is written in the body of the `compute` method. This method is called whenever an input event is available, and a new output event is required. Its first argument is an array of Java objects; the size of that array is that of the input arity we declared for this processor (in our case: 1). Computing the length amounts to extracting the first (and only) event of array inputs, casting it to a String, and getting its length. The end result is this:

```
public class StringLength extends SynchronousProcessor
{
  public StringLength()
  {
    super(1, 1);
  }

  @Override
  public boolean compute(Object[] inputs, Queue<Object[]> outputs)
  {
    int length = ((String) inputs[0]).length();
    return outputs.add(new Object[]{length});
  }

  @Override
  public Processor duplicate(boolean with_state)
  {
    return new StringLength();
  }
```

```java
  @Override
  public void getInputTypesFor(Set<Class<?>> classes, int position)
  {
    if (position == 0)
    {
      classes.add(String.class);
    }
  }

  @Override
  public Class<?> getOutputType(int position)
  {
    if (position == 0)
    {
      return Number.class;
    }
    return null;
  }
}
```

✎

Note that the `compute` method has a second argument, which is a queue of object arrays. If the processor is of arity n, it must create an event front of size n, which means putting an event into each of its n output queues. It may also decide to output more than one such n-uplet for a single input event, and these events are accumulated into a queue –hence the slightly odd object type. The method puts into the `outputs` queue an array of `Objects` with a single element, which, in this case, is an integer corresponding to the input string's length.

The return type of the `compute` method is a `boolean`. This value is used to signal to a `Pullable` object whether the processor is expected to produce more events in the future. A processor should return `false` only if it is absolutely sure that no more events will be produced in the future; in all other situations, it must return `true`. Examples of processors whose `compute` method returns `false` are processors that read from a file; when the end of the file is reached, they return false to indicate that no more new events are expected. Except in very special situations such as these, a processor should return `true`.

The other method that needs to be implemented is `duplicate`; it works in the

same way as for functions, and in general only consists of returning a new instance of the class. However, the reader should notice that for processors, `duplicate` takes a Boolean argument, called `with_state`. If this argument is set to `true`, the processor should not simply create a new copy of itself; it must also transfer its current *state* to the new object. Typically, this means that if the processor has member fields that determine its behaviour, these member fields must be set to the same values in the newly created copy. This is not the case in this simple example, since the processor we create has no member field at all. Therefore, method `duplicate` simply ignores the argument and returns a new instance of `StringLength`. From then on, a user can instantiate `StringLength`, connect it to the output of any other processor that produces strings, and pipe its result to the input of any other processor that accepts numbers.

Optionally, a processor can declare its input and output types, as is the case for function objects. Therefore, one can override the methods `getInputTypesFor` and `getOutputType`. In the present case, the type of the first (and only) input stream is `String`, while the type of the first (and only) output stream is `Number`. This leads to the methods shown in the previous code snippet. If a processor does not override these methods, by default the `Processor` class returns a special type called `Variant`. The occurrence of such a type in an input or output pipe disables the type checking step that the `Connector` class normally performs before connecting two processors together.

GREATER INPUT AND OUTPUT ARITY

This second example displays a processor taking two traces as input. The events of each trace are instances of a simple user-defined class called `Point`, defined as follows:

```
public class Point {
  public float x;
  public float y;
}
```

We will write a processor taking one event (i.e. one Point) from each input trace, and return the Euclidean distance between these two points.

```
public class EuclideanDistance extends SynchronousProcessor
{
  public static final EuclideanDistance instance =
```

```
      new EuclideanDistance();

  EuclideanDistance()
  {
    super(2, 1);
  }

  public boolean compute(Object[] inputs, Queue<Object[]> outputs)
  {
    Point p1 = (Point) inputs[0];
    Point p2 = (Point) inputs[1];
    double distance = Math.sqrt(Math.pow(p2.x - p1.x, 2)
        + Math.pow(p2.y - p1.y, 2));
    outputs.add(new Object[] {distance});
    return true;
  }

  @Override
  public Processor duplicate(boolean with_state)
  {
    return this;
  }
}
```

✎

As the reader can see, the `compute` method now expects the `input` array to contain *two* elements, and these two elements are cast as instances of `Point`. In addition, the `duplicate` method introduces a small variant: rather than returning a new copy of the processor, it returns itself (`this`). This behaviour makes sense in the case of a **singleton** –that is, an object which exists in a single copy across an entire program. In such a case, a good practice is to reduce the visibility of the class' constructor (to prevent users from calling it and creating new instances), and to provide instead a static reference to a single instance of the object. This is the goal of the `instance` static field.

As we have seen in earlier chapters, many BeepBeep objects (especially functions) are singletons. For example, the utility classes `Booleans` and `Numbers` provide static references to a few general-purpose objects, such as `Booleans.and` or `Numbers.addition`. Singleton processors are less common, but this example shows that it is possible to implement them in a clean way.

As we know, a processor is not limited to producing a single output stream. In

this example, we show how to implement a processor with an output arity of two. This processor takes as input a single trace of `Points` (see the example above), and sends the x and y component of that point as events of two output streams.

```
public class SplitPoint extends SynchronousProcessor
{
  public SplitPoint()
  {
    super(1, 2);
  }

  @Override
  protected boolean compute(Object[] inputs, Queue<Object[]> outputs)
  {
    Point p = (Point) inputs[0];
    outputs.add(new Object[] {p.x, p.y});
    return true;
  }

  @Override
  public Processor duplicate(boolean with_state)
  {
    return new SplitPoint();
  }
}
```

✎

This time, the processor adds to the output queue an array of size 2. One must remember that it is an error to add an array whose size is not equal to the processor's output arity. Although this may not be detected immediately, such an incorrect behaviour is likely to create exceptions at some point in the execution of the program.

NON-UNIFORM PROCESSORS

So far, all processors that were designed return one output event for every input event (or pair of events) they receive. (As a matter of fact, it would have been easier to implement them as `Functions` that could have been passed to an `ApplyFunction` processor.) In BeepBeep's terminology, these processors

are called **uniform** (or more precisely, 1-uniform). However, this needs not to be the case. The following processor outputs an event if its value is greater than 0, and no event at all otherwise.

```
public class OutIfPositive extends SynchronousProcessor {

    public OutIfPositive() {
        super(1, 1);
    }
```

✎

The way to indicate that a processor does not produce any output for an input is to simply add nothing to the output queue. Note that this should not be confused with returning `false`, which signifies that the processor will never output any event in the future.

Conversely, a processor does not need to output only one event for each input event. For example, the following processor repeats an input event as many times as its numerical value: if the event is the value 3, it is repeated 3 times in the output. Method `compute` of this processor would look like the following:

```
public boolean compute(Object[] inputs, Queue<Object[]> outputs)
{
  System.out.println("Call to compute");
    Number n = (Number) inputs[0];
    for (int i = 0; i < n.intValue(); i++)
    {
        outputs.add(new Object[] {inputs[0]});
    }
    return true;
}
```

✎

This example shows why `compute`'s `outputs` argument is a *queue* of arrays of objects. In this class, a call to `compute` may result in more than one event being output. If `compute` could output only one event at a time, our processor would need to buffer the events to output somewhere, and draw events from that buffer on subsequent calls to `compute`. Fortunately, the `SynchronousProcessor` class handles this in a transparent manner. Therefore, `compute` can put as many events as one wishes in the output queue, and `SynchronousProcessor` is responsible for releasing them one by one through its `Pullable` object.

This example puts in light an interesting feature of the `SynchronousProcessor` class. Notice how we inserted a `println` statement in the first line of method `compute`. This allows us to track the moments where method `compute` is being called by BeepBeep. Consider the following program:

```
QueueSource src = new QueueSource();
src.setEvents(1, 2, 1);
Stuttering s = new Stuttering();
Connector.connect(src, s);
Pullable p = s.getPullableOutput();
for (int i = 0; i < 4; i++)
{
  System.out.println("Call to pull: " + p.pull());
}
```

✎

This program calls `pull` on an instance of the `Stuttering` processor four times. The first call to `pull` triggers a call to `compute` on `s` in the background; this explains the first two lines printed at the console:

```
Call to compute
Call to pull: 1
```

The second call to `pull` results in another call to `compute` on `s`, producing the value 2, and printing the next two lines:

```
Call to compute
Call to pull: 2
```

However, the third call to `pull` directly outputs the value 2, without triggering a call to `compute`; therefore, the next line to be printed is:

```
Call to pull: 2
```

This may seem surprising, but can easily be explained. The previous call to `pull` made `s` receive the input event 2. As a result, the call to `compute` put into the `outputs` queue *two* object arrays, each containing the number 2. The first object array was immediately retrieved and returned as the result of the call to `pull`, while the contents of the second object array was put into the processor's output queue, waiting for the next call to `pull`. Consequently, upon the next call to `pull`, there was no need to call `compute`, since an output event was already waiting in the processor's output queue, ready to be retrieved.

Therefore, when designing a new `SynchronousProcessor`, one must keep in

mind that calls to `push` (resp. `pull`) on a processor's `Pushable` (resp. `Pullable`) do not always correspond to a call to `compute`, depending on the current contents of the processor's input and output queues.

STATEFUL PROCESSORS

So far, all our processors are "memoryless": they keep no information about past events when making their computation. It is also possible to create "memoryful" processors. As an example, let us create a processor called `MyMax`, which outputs the maximum between the current event and the previous one. Given the following input trace:

```
5, 1, 2, 3, 6, 4, ...
```

...the processor should output:

```
(nothing), 5, 2, 3, 6, 6, ...
```

Notice how, after receiving the first event, the processor should not return anything yet, as two events are needed before being able to output something. A possible implementation could be the following:

```
public class MyMax extends SynchronousProcessor
{
  Number last = null;

  public MyMax()
  {
    super(1, 1);
  }

  @Override
  public boolean compute(Object[] inputs, Queue<Object[]> outputs)
  {
    Number current = (Number) inputs[0];
    Number output;
    if (last != null)
    {
      output = Math.max(last.floatValue(), current.floatValue());
      last = current;
      outputs.add(new Object[]{output});
    }
    else
```

```java
    {
      last = current;
    }
    return true;
  }

  @Override
  public Processor duplicate(boolean with_state)
  {
    MyMax mm = new MyMax();
    if (with_state)
    {
      mm.last = this.last;
    }
    return mm;
  }
}
```

✎

This processor is the first in this chapter to have a member field, called `last`. When the processor is instantiated, `last` is set to `null`. Each call to `compute` compares the value of `last` with the current event (if `last` is not null), and then sets the value of `last` to the current event. Therefore, the member field `last` acts as a form of "memory": for a given input event, the processor will produce a different output depending on the contents of this field –which itself depends on the previous event given to the processor.

The presence of a member field changes the way of implementing method `duplicate`. Remember that a processor has the option of being copied *along with its state*, by setting the value of argument `with_state` to `true`. Therefore, the code for `duplicate` must take into account this additional possibility. Notice how a new instance of `MyMax`, called `mm`, is created; if the duplication is stateful, an extra step is taken to copy the current value of `last` into `mm`. This has for effect of putting `mm` into the same state as the current object.

The implementation of `duplicate` is probably the most delicate part in the creation of a new stateful `SynchronousProcessor`. Failing to create a faithful copy of the original object (for example, by failing to transfer the values of all the appropriate member fields) may result in unforeseen and hard-to-debug behaviours. As an example, let us go back to the `Stuttering` processor we created previously. Consider the following program:

```
QueueSource src1 = new QueueSource();
src1.setEvents(2, 1);
Stuttering s1 = new Stuttering();
Connector.connect(src1, s1);
Pullable p1 = s1.getPullableOutput();
System.out.println("Call to pull on p1: " + p1.pull());
```

✎

```
Call to compute
Call to pull on p1: 2
```

Let us now make a stateful duplicate of `s1`, connect it to a new event source, and call `pull` once:

```
QueueSource src2 = new QueueSource();
src2.setEvents(3, 1);
Stuttering s2 = s1.duplicate(true);
Connector.connect(src2, s2);
Pullable p2 = s2.getPullableOutput();
System.out.println("Call to pull on p2: " + p2.pull());
```

✎

The program prints:

```
Call to compute
Call to pull on p2: 3
```

However, this is not the expected output. As we have seen in a previous example, the next event that should be output by processor `s1` is the second instance of number 2. Processor `s2` should be a *stateful* copy of `s1`, and hence, produce the same output event. Instead, the call to `pull` on `s2` resulted in `s2` pulling number 3 from `src2` and sending it to its output. The reason for this strange behaviour is the fact that, when `s1` was duplicated, the contents of its input and output queues was not transferred to `s2`. It turns out that the events present in these queues, most of the time, are also part of a processor's state.

Technically, a descendent of `SynchronousProcessor` does not have direct access to these queues. Rather than copying their contents manually in every implementation of `duplicate`, a shortcut consists of calling a method called `duplicateInto`, provided by the `Processor` class. This method receives as an argument the target copy of the `Processor`; it is responsible for copying the

contents of the input and output queues into this object. We can hence create a new version of `Stutter`, called `StutteringCopy`, containing a "corrected" version of method `duplicate`. The method now looks as follows:

```
public StutteringCopy duplicate(boolean with_state)
{
  StutteringCopy s = new StutteringCopy();
  if (with_state)
  {
    super.duplicateInto(s);
  }
  return s;
}
```

✎

We can now write a new version of our program, which uses the `Stuttering-Copy` object in place of `Stuttering`:

```
QueueSource src1 = new QueueSource();
src1.setEvents(2, 1);
StutteringCopy s1 = new StutteringCopy();
Connector.connect(src1, s1);
Pullable p1 = s1.getPullableOutput();
System.out.println("Call to pull on p1: " + p1.pull());
QueueSource src2 = new QueueSource();
src2.setEvents(3, 1);
StutteringCopy s2 = s1.duplicate(true);
Connector.connect(src2, s2);
Pullable p2 = s2.getPullableOutput();
System.out.println("Call to pull on p2: " + p2.pull());
```

✎

This time, the program prints, as expected:

```
Call to compute
Call to pull on p1: 2
Call to pull on p2: 2
```

In this chapter, we have seen how BeepBeep's functionalities can be extended by letting users invent their own `Processor` and `Function` objects. All of Beep-Beep's palettes are created in this way: a palette is just a pre-compiled JAR

bundle of classes that depend on BeepBeep's core (and possibly other external libraries). We strongly encourage the reader to experiment with creating new processors specific to the use cases they may encounter. It is hoped that BeepBeep's palette architecture, combined with its simple extension mechanisms, will help third-party users contribute to the BeepBeep ecosystem by developing and distributing extensions suited to their own needs.

8 Designing a Query Language

In this chapter, we shall explore a unique feature of BeepBeep, which is the possibility to create custom **query languages**. Rather than instantiate and pipe processors directly through Java code, a query language allows a user to create processor chains by writing expressions using a custom syntax, effectively enabling the creation of **domain-specific languages** (DSLs).

The Turing tarpit of a Single Language

As already mentioned at the very beginning of this book, many other event stream processing engines provide the user with their own query language. In most of these systems, the syntax for these languages is borrowed from SQL, and many stream processing operations can be accomplished by writing statements such as `SELECT`. For example, the following query, taken from the documentation of a CEP system called Esper, selects a total price per customer over pairs of events (a ServiceOrder followed by a ProductOrder event for the same customer id within one minute), occurring in the last two hours, in which the sum of price is greater than 100, and using a *where* clause to filter on the customer's name:

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
  b=ProductOrder(custId = a.custId)
where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

In the field of runtime verification, the majority of tools rather use variants of languages closer to mathematical logic or finite-state machines.

For example, the following property, expressed in a language called MFOTL used by the MonPoly tool, checks that for each user, the number of withdrawal peaks in the last 31 days does not exceed a threshold of five, where a withdrawal peak is a value at least twice the average over the last 31 days:

```
□ ∀*u*:∀*c*: [CNT~j~ v; p; κ : [AVG ~a~ a;τ.⧫~[0;31)~
withdraw(*u*; a) ∧ ts(τ)](v; *u*) ∧
⧫~[0;31)~ withdraw(*u*; p) ∧ ts(κ) ∧ 2 · V ≺ p](*c*; *u*) → c ⪯ 5
```

The main problem with all of these systems is that they force the user to use them through their query language exclusively. Contrary to BeepBeep, one seldom has direct access to the underlying objects performing the computations. Most importantly, as each of these systems aim to be versatile and applicable to a wide variety of problems, their query language becomes extremely complex: every possible operation on streams has to be written as an expression of the single query language they provide. A typical symptom of this, in some CEP systems, is the presence of tentacular `SELECT` statements with a dozen optional clauses attempting to cover every possible case. Runtime verification tools fare no better on this respect, and complex nested logical expressions of multiple lines regularly show up in research papers about them. In all cases, the legibility of the resulting expressions suffers a lot. Although there is almost always a way to twist a problem so that it can fit inside any system's language *theoretically*, in practice many such expressions are often plain unusable. This can arguably fall into the category of what computer scientist Alan Perlis has described as a "Turing tarpit":

> Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

In contrast, BeepBeep was designed based on the observation that no single language could accommodate every conceivable problem on streams –at least in a simple and intuitive way. Rather than trying to design a "one-size-fits-all" language, and falling victim to the same problem as other systems, BeepBeep provides no built-in query language at all. Instead, it offers users the possibility to easily create their own query languages, using the syntax they wish, and including only the features they need.

The basic process of creating a DSL is as follows:

1. First, users decide what expressions of the language will look like by defining what is called a *grammar*
2. Then, users devise a mechanism to build objects (typically `Function`

and `Processor` objects) from expressions of the language

Defining a Grammar

A special palette called `dsl` allows the user to design query languages for various purposes. Under the hood, `dsl` uses Bullwinkle, a parser for languages that operates through recursive descent with backtracking. Typical parser generators such as ANTLR, Yacc or Bison take a grammar as input and produce code for a parser specific to that grammar, which must then be compiled to be used. On the contrary, Bullwinkle reads the definition of a grammar at *runtime* and can parse strings on the spot.

The first step in creating a language is therefore to define its **grammar**, i.e. the concrete rules that define how valid expressions can be created. This can be done by pasing a character string (taken from a file or created directly) containing the grammar declaration. Here is a very simple example of such a declaration:

```
<exp> := <add> | <sbt> | <num> ;
<add> := <num> + <num> ;
<sbt> := <num> - <num> ;
<num> := 0 | 1 | 2 ;
```

The definition of the grammar must follow a well-known notation called Backus-Naur Form (BNF). In this notation, the grammar is defined as a series of **rules** (one rule per line). The part of the rule at the left of the `:=` character contains exactly one **non-terminal symbol**. The right-hand side of the rule contains one or more **cases**, separated by the pipe (`|`) character. Each case is a sequence made of literals (character strings to be interpreted literally) and non-terminal symbols. The first non-terminal appearing in the grammar has a special meaning, and is called the **start symbol**.

Taken together, the rules define a set of expressions called *valid* expressions. In the above example, this specific grammar defines a simple subset of arithmetical expressions, involving only addition, subtraction, and three numbers. An expression is valid if there exists a way to begin at the start symbol, and successively apply rules from the grammar to ultimately produce that expression.

According to the grammar above, the expression `1 + 0` is valid, since it is

possible to begin at the start symbol <exp> and apply rules to obtain the expression. An algorithm could perform the following manipulations:

1. Transform <exp> into <add> according to the first case of rule 1.
2. Transform <add> into <num> + <num> according to the (only) case of rule 2.
3. Transform the first <num> into 1 according to the second case of rule 4; the expression becomes 1 + <num>.
4. Transform the second <num> into 0 according to the first case of rule 4; the expression becomes 1 + 0.

On the contrary, the expression 1 + 0 - 2 is not valid as there is no possible way to apply the rules in the grammar to transform <exp> into that expression.

To define a grammar from a set of BNF rules, a few conventions must be followed. First, non-terminal symbols are enclosed in < and > and their names must not contain spaces. As seen previously, rules are defined with := and cases are separated by the pipe character. A rule may span multiple lines (any whitespace character after the first one is ignored, as in e.g. HTML) and must end by a semicolon.

In the previous example, the grammar can accommodate only the numbers 0 to 2. Since Bullwinkle only accepts the terminal symbols explicitly written into the grammar, as many cases for <num> would need to be written as there are integers, which is not very practical. Fortunately, terminal symbols can also be defined through *regular expressions*. A regular expression (regex for short) describes a pattern of characters. Regex terminals are identified with the ^ (hat) character. For example, to indicate that any string of one or more digits is accepted, one could rewrite the rule for <num> as follows:

```
<num> := ^[0-9]+;
```

The expression [0-9]+ is a regex pattern; here, it designates any string of numbers. Explaining regular expressions is beyond the scope of this chapter. The reader is referred to the very large documentation on the topic available in books and online.

A BNF grammar can also be *recursive*; that is, a rule <A> can contain a case that involves the non-terminal <B>, which itself can have a case that refers to <A>. One can rewrite the original grammar in a slightly more complex way, such that nested operations are allowed:

```
<exp> := <add> | <sbt> | <num> ;
```

```
<add> := ( <exp> ) + ( <exp> ) ;
<sbt> := ( <exp> ) - ( <exp> ) ;
<num> := ^[0-9]+;
```

Note how the operands for `<add>` and `<sbt>` involve the non-terminal `<exp>`. Using such a grammar, an expression like `(3)+((4)-(5))` is valid. However, according to the rules, the use of parentheses is mandatory, even around single numbers. This can be relaxed by adding further cases to `<add>` and `<sbt>`, which become:

```
<add> := <num> + <num> | <num> + ( <exp> )
         | ( <exp> ) + <num> | ( <exp> ) + ( <exp> );
<sbt> := <num> - <num> | <num> - ( <exp> )
         | ( <exp> ) - <num> | ( <exp> ) - ( <exp> );
```

With this new grammar, it is now possible to write a more natural expression such as `3+(4-5)`.

The Bullwinkle parser offers many more features, which shall not be discussed here. For example, it accepts a second way of defining a grammar by assembling rules and creating instances of objects programmatically; we refer the reader to the online documentation for more details. A final remark regarding grammars is that they must belong to a special family called LL(k). Roughly, this means that they must not contain a production rules of the form `<S> := <S> something`. Trying to parse such a rule by recursive descent (the algorithm used by Bullwinkle) causes an infinite recursion (which will throw a `ParseException` when the maximum recursion depth is reached).

From a grammar defined as above, one can create an instance of an object called a `BnfParser`. For example, suppose that the grammar for arithmetical expressions is contained in a text file called `arithmetic.bnf`. Obtaining a parser for that object can be done as follows:

```
InputStream is = ParserExample.class
    .getResourceAsStream("arithmetic.bnf");
BnfParser parser = new BnfParser(is);
ParseNode root = parser.parse("3 + (4 - 5)");
```

✎

Once a grammar has been loaded into an instance of `BnfParser`, it is possible to read character strings through its `parse()` method. This is what is done the last instruction above: the string `3+(4-5)` is passed to `parse`, and the method returns an object of type `ParseNode`. This object corresponds to the root of a

structure called a **parsing tree**. The tree follows the structure of the parsed expression, and specifies how it can be derived from the start symbol using the rules defined by the grammar. The parsing tree for the expression `3+(4-5)` looks like this:

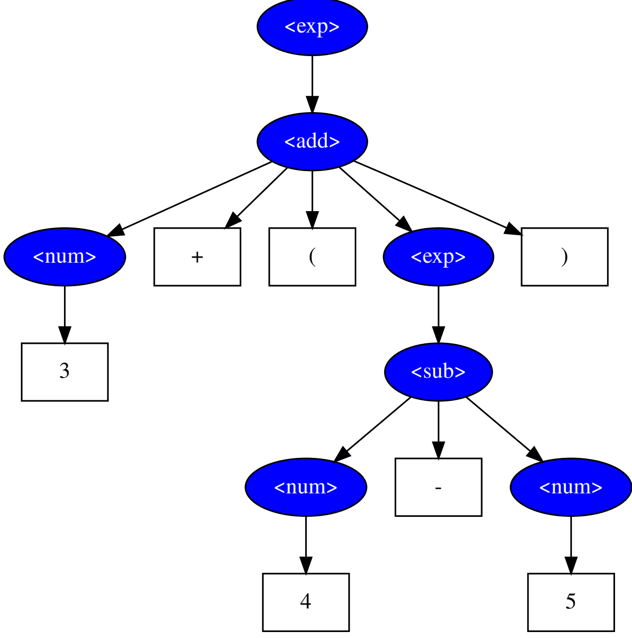


Figure 8.1: The parsing tree for the expression `3+(4-5)`.

The leaves of this tree are literals; all the other nodes correspond to non-terminal symbols. Intuitively, a node represents the application of a rule, and the children of that node are the symbols in the specific case of the rule that was applied. For example, the root of the tree corresponds to the start symbol `<exp>`; this symbol is transformed into `<add>` by applying the first case of rule 1. The symbol `add`, in turn, is transformed into the expression `<num> + ( <exp> )` by applying the second case of rule 2 –and so on.

Building Objects from the Parsing Tree

As we can see, the process of parsing transforms an arbitrary character string into a structured tree. Using this tree to construct an object is much easier

than trying to process a character string directly: one simply needs to traverse the parsing tree, and to build the parts of the object piece by piece. This is done by using an object called the `GrammarObjectBuilder`.

To illustrate the principle, consider this simple grammar to represent arithmetic expressions in Polish notation, such as this:

```
<exp> := <add> | <sbt> | <num>;
<add> := + <exp> <exp>;
<sbt> := - <exp> <exp>;
<num> := ^[0-9]+;
```

Using such a grammar, the expression `3+(4-5)` is written as `+ 3 - 4 5`. The goal is to create a `FunctionTree` object from expressions following this syntax.

The first step is to create a new empty class that extends `GrammarObjectBuilder`. The constructor of this class should call a method called `setGrammar()`, and pass a string containing the BNF grammar corresponding to the language.

```
public ArithmeticBuilder()
{
    super();
    try
    {
        setGrammar("<exp> := <add> | <sbt> | <num>;\n"
                + "<add> := + <exp> <exp>;\n"
                + "<sbt> := - <exp> <exp>;\n"
                + "<num> := ^[0-9]+;");
    }
    catch (InvalidGrammarException e)
    {
    }
}
```

✎

The `GrammarObjectBuilder` class defines a method called `build()`, which takes a character string as input. It first parses that string, and then performs a *postfix* traversal of the resulting parsing tree, maintaining in its memory a stack of arbitrary objects along the way. A postfix traversal implies that the nodes of the tree are visited one by one; furthermore, before a parent node is visited, all its children are visited first. Hence, in the tree shown above, the first node to be visited will be the leftmost number `3`, followed by its parent

<num>, and so on.

The `GrammarObjectBuilder` treats any terminal symbol as a character string. Therefore, when visiting a leaf of the parsing tree, `GrammarObjectBuilder` puts on its stack a `String` object whose value is the contents of that specific literal. When visiting a parse node corresponding to a non-terminal token, such as <add>, the builder seeks a method that handles this symbol. “Handling” a symbol generally amounts to popping objects from the stack, creating one or more new objects, and pushing these objects back onto the stack. Therefore, to build a `FunctionTree` from an expression, the `ArithmeticBuilder` class must define methods that take care of each non-terminal symbol in the grammar we defined.

Let us begin with the simplest case, the <num> symbol. When a <num> node is visited in the parsing tree, according to the postfix traversal we described earlier, we know that the top of the stack contains a string with the number that was parsed. The task of the method is to take this string, convert it into a Java `Number` object, and create a BeepBeep `Constant` object from this number. Therefore, a method called `handleNum` can be created, as in the following code snippet:

```
public void handleNum(ArrayDeque<Object> stack)
{
    String s_num = (String) stack.pop();
    Number n_num = Float.parseFloat(s_num);
    Constant c = new Constant(n_num);
    stack.push(c);
}
```

✎

As you may have noticed, this method receives as an argument the current contents of the object stack maintained by the `GrammarObjectBuilder` object. It is up to each method to pop and push objects from the stack, in order to recursively create the desired object at the end. This process can also be illustrated graphically, as in Figure 8.2.

To the left-hand side of the diagram, a box represents the top of the object stack when the method is called. Here, the pattern expects the stack to contain a String object with a numerical value *n*. The right-hand side of the stack represent the content of the object stack after the method returns. Here, one can see that the string at top of the stack has been popped, and replaced by a

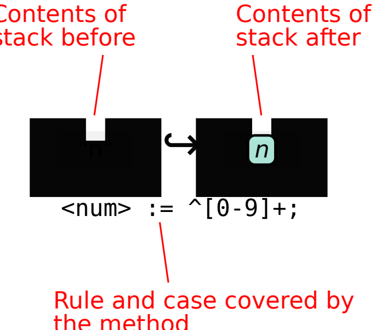


Figure 8.2: A graphical representation of the stack manipulations for rule `<num>`.

`Constant` object with the value *n*. The stack may contain other objects below, but they are not relevant to the application of this method. For the sake of clarity, the grammar rule and case corresponding to this operation are often written next to the diagram.

What remains to be done is to report to the object builder that this method should be called whenever a `<num>` tree node is visited. This can be done by adding an annotation `@Builds` to the method, which reads as follows:

```
@Builds(rule="<num>")
```

Users should place this annotation just above the first line that declares the method signature. The operation of this method can also be illustrated graphically as in the following figure.

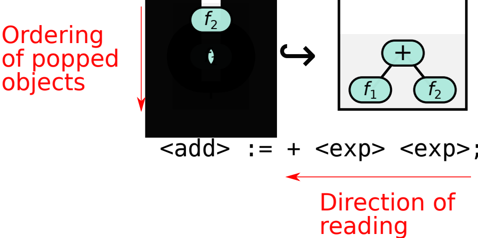
Let us now have a look at the code to handle token `add`.

```
@Builds(rule="<add>")
public void handleAdd(ArrayDeque<Object> stack)
{
    Function f2 = (Function) stack.pop();
    Function f1 = (Function) stack.pop();
    stack.pop();
    stack.push(new FunctionTree(Numbers.addition, f1, f2));
}
```

✎

When a parse node for `add` is visited, the object stack should already contain the `Function` objects created from its two operands, since the builder traverses

the tree in a postfix fashion. This is illustrated by the following diagram:



and using the resulting `Function` object can be achieved in a few lines, as the code below illustrates.

```
ArithmeticBuilder builder = new ArithmeticBuilder();
Function f = builder.build("+ 3 - 4 5");
Object[] value = new Object[1];
f.evaluate(new Object[]{}, value);
System.out.println(value[0]);
```

✎

The first instruction creates a new instance of the `ArithmeticBuilder`. The second calls the `build` method on the string `+ 3 - 4 5`. Since `ArithmeticBuilder` is parameterized with the type `Function`, the return value of `build`, `f`, is correctly cast as a `Function` object. The remaining lines simply prepare a call to `evaluate` on `f` and print its return value. This function contains no `StreamVariables`, takeing no argument as its input. The end result, printed at the console, is indeed the value of `3+(4-5)`:

```
2.0
```

As a matter of fact, a simple calculator that can read strings in Polish notation and compute their value has just been written. This was done using BeepBeep's `Function` objects, a simple grammar and a custom-built `GrammarObjectBuilder`. So far, only 4 lines of text were required for the grammar, and about 20 lines of code for the interpreter. Just for fun, this can even be turned into an interactive command line tool, as follows:

```
Scanner scanner = new Scanner(System.in);
ArithmeticBuilder builder = new ArithmeticBuilder();
while (true)
{
    System.out.print("? ");
    String line = scanner.nextLine();
    if (line.equalsIgnoreCase("q"))
        break;
    Function f = builder.build(line);
    Object[] value = new Object[1];
    f.evaluate(new Object[]{}, value);
    System.out.println(value[0]);
}
scanner.close();
```

✎

This program simply reads expressions at the console, parses and evaluates them, and prints their result until the user writes q:

```
? + 2 3
5.0
? - 5 + 4 4
-3.0
? q
```

Simpler Stack Manipulations

As one can see, it is possible to create builders that read expressions and create new objects with very little effort. However, the manipulation of the stack in each method remains a delicate operation. Popping one object less than expected, or one more, may put the stack in an inconsistent state and have disastrous cascading effects on the build process. As a simple example, suppose that method handleAdd is modified as follows:

```
@Builds(rule="<add>")
public void handleAdd(ArrayDeque<Object> stack)
{
    Function f2 = (Function) stack.pop();
    stack.pop();
    Function f1 = (Function) stack.pop();
    stack.push(new FunctionTree(Numbers.addition, f1, f2));
}
```

✎

The last two calls to pop are simply swapped, implying that the second object on the stack is now discarded, while the first and third are cast as Function objects. Trying to run this modified program will produce a screenful of exceptions:

```
Exception in thread "main"
    at ca.uqac.lif.bullwinkle.ParseTreeObjectBuilder.build
    at ca.uqac.lif.cep.dsl.GrammarObjectBuilder.build
    at dsl.ArithmeticBuilderIncorrect.main
Caused by:
    at ca.uqac.lif.bullwinkle.ParseTreeObjectBuilder.visit
    at ca.uqac.lif.bullwinkle.ParseNode.postfixAccept
    ...
```

As a result, one has to be very careful when interacting with the object stack. However, it turns out that in many cases, a user does not need to manipulate this stack directly. Looking back at the `ArithmeticBuilder` written earlier, one notices that every method actually does the same thing:

- It pops as many objects from the stack as there are tokens in the corresponding grammar rule, in reverse from the order they appear in the rule.
- It instantiates a new object by using elements that were popped from the stack.
- It puts that new object back onto the stack.

It is possible to instruct the object builder to automate this repetitive process, using an additional argument to the `@Builds` annotation called `pop`. For example, the annotation for the `<num>` symbol would now read:

```
@Builds(rule="<num>", pop=true)
```

The use of `pop` also changes the signature of our handler method, which becomes:

```
@Builds(rule="<num>", pop=true)
public Constant handleNum(Object ... parts)
{
    return new Constant(Float.parseFloat((String) parts[0]));
}
```

✎

First, one should notice that the method no longer receives a stack as an argument, but rather an array of objects called `parts`. The use of `pop` instructs the builder to already pop the appropriate number of objects from the stack, based on the number of tokens in the corresponding rule of the grammar. Here, the rule for `<num>` has a single token, which is a string of digits. Therefore, the array `parts` will contain a single `String` object at index 0.

The second observation is that the method now returns something. The return value should correspond to the object that should be put back onto the stack at the end of the operation. In the case of `<num>`, the return value is a `Constant` object created by extracting a `Float` from the string received from `parts`. Notice how the original 4-line method has been simplified to a single instruction. Moreover, we no longer have to manually pop and push objects onto the stack: the object builder takes care of this outside of our handler

method. This reduces the amount of work required, but also the possibility of making mistakes.

Similarly, a handler method for <add> would look like this:

```
@Builds(rule="<add>", pop=true)
public FunctionTree handleAdd(Object ... parts)
{
    return new FunctionTree(Numbers.addition,
            (Function) parts[1], (Function) parts[2]);
}
```

✎

The rule for <add> has three tokens. Based on that rule, the contents of `parts` will be made of three objects: the first is the "+" string; the other two are the `Function` objects that are the operands of the addition. We know that, by the time this method is called, these two functions have already been created by the previous building steps and placed on the stack. Again, notice how the five lines of the original method have been replaced by a single instruction.

Now, consider a more complex grammar, this time defining arithmetic operations using the more natural *infix* notation.

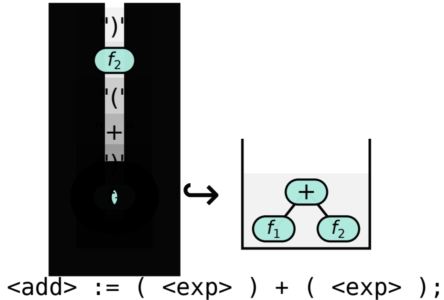
```
<exp> := <add> | <sbt> | <num> ;
<add> := <num> + <num> | <num> + ( <exp> )
           | ( <exp> ) + <num> | ( <exp> ) + ( <exp> );
<sbt> := <num> - <num> | <num> - ( <exp> )
           | ( <exp> ) - <num> | ( <exp> ) - ( <exp> );
<num> := ^[0-9]+
```

This time, the rules for each operator must take into account whether any of their operands is a number or a compound expression. Writing an object builder for this grammar is slightly more complex. The handler methods for <add> and <sbt> now have multiple cases; these cases do not have the same number of operands, and the position of the <exp> operands among the tokens for each case is not always the same. Therefore, one would have to carefully pop an element, check if it is a parenthesis, and if so, take care of popping the matching parenthesis later on, and so on. This is perfectly possible, although a little tedious:

```
public ArithExp handleAdd(Object ... parts)
{
    Function left, right;
```

```java
    int index ;
    if (parts[0] instanceof String) {
        left = (Function) parts[1];
        index = 4;
    }
    else {
        left = (Function) parts[0];
        index = 2;
    }
    if (parts[index] instanceof String)
        right = (Function) parts[index + 1];
    else
        right = (Function) parts[index];
    return new FunctionTree(Addition.instance, left, right);
}
```

Notice how one must first check if the first object in `parts` is a string (corresponding to an opening parenthesis); if so, the first operand is located at index 1, otherwise it is at index 0. This, in turn, shifts the index of the second operand, which may or may not be surrounded by parentheses. The case where both operands are between parentheses could be illustrated as follows:



```
@Builds(rule="<add>", pop=true, clean=true)
public FunctionTree handleAdd(Object ... parts) {
  return new FunctionTree(Addition.instance,
    (Function) parts[0], (Function) parts[1]);
}
```

The array indices become 0 and 1, since only the two `FunctionTree` objects remain as arguments. This results in the picture below. Notice how the non-terminal symbols `<exp>` in the rule are underlined, to emphasize the fact that they are the only symbols to be represented on the object stack at the right; the interspersed terminal tokens between these symbols are not shown.

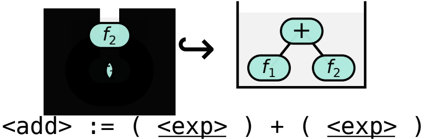


Figure 8.6: A graphical representation of the stack manipulations for rule `<add>` in infix notation, using the `clean` option.

Building Processor Chains

So far, the examples have focused on simple grammars building `Function` objects in various ways. The process for building and chaining `Processor` objects is largely similar; however, since processors must be connected to each other in a specific way, one will need to pay attention to this detail when manipulating these objects on the stack.

As a simple example, we shall illustrate how a small language can be used to chain processors from BeepBeep's core. Let us start with the grammar. We will focus on a handful of basic processors, namely `Trim`, `CountDecimate` and `Filter`. For each of them, a simple syntax is defined to use them. The grammar could then look as follows:

```
<proc>   := <trim> | <decim> | <filter> | <stream> ;
<trim>   := TRIM <num> FROM ( <proc> );
<decim>  := KEEP ONE EVERY <num> FROM ( <proc> );
<filter> := FILTER ( <proc> ) WITH ( <proc> );
<stream> := INPUT <num> ;
<num>    := ^[0-9]+;
```

The start symbol of the grammar is <proc>, which itself can be one of four different cases. The <stream> construct is used to designate the input pipes of the resulting processor; as a processor chain can have multiple inputs, the number of the corresponding input must be mentioned in the construct.

Let us now examine the code handling each rule one by one, starting with the rule for <trim>:

```
@Builds(rule="<trim>", pop=true, clean=true)
public Trim handleTrim(Object ... parts)
{
    Integer n = Integer.parseInt((String) parts[0]);
    Processor p = (Processor) parts[1];
    Trim trim = new Trim(n);
    Connector.connect(p, trim);
    add(trim);
    return trim;
}
```

✎

According to the grammar rule for <trim>, the contents of the `parts` array should be a string of digits, and an instance of a `Processor` object. The first two instructions retrieve these two objects. The third instruction instantiates a new `Trim` processor by using the parsed integer for the number of elements to trim. The processor passed as an argument is connected to the newly created `trim` processor, and `trim` is returned onto the object stack.

The second-last instruction warrants an explanation. The goal of the `Group-ProcessorBuilder` is to ultimately return a `GroupProcessor` whose contents are made of the processors instantiated and connected during the building process. However, in order for these objects to be added to the resulting `GroupProcessor`, the `GroupProcessorBuilder` needs to be notified that these objects are created. This is the purpose of the call to the `add` method.

This whole process can be represented as in Figure 8.7.

This illustration stipulates that an arbitrary processor P and a string “n” are popped from the stack; a new `Trim(n)` processor is created and connected to the end of P; finally, this `Trim` processor is pushed back on the stack. Notice how, in this diagram, processor P seems to hang outside of the stack on the right-hand side of the picture. This is due to the fact that at the end of the operation, only the `Trim` processor is at the top of the stack; the reference to

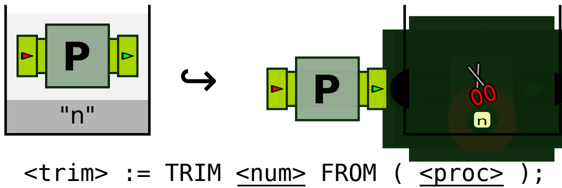


Figure 8.7: A graphical representation of the stack manipulations for rule `<trim>`.

processor P is no longer present there. Yes, P is *connected* to `Trim`, but this only means that the respective pullables and pushables of both processors are made aware of each other. To illustrate this, P is drawn outside of the stack, but shown piped to the processor that is on the stack.

Once this is understood, the code for rule `<decim>` is straightforward, and almost identical to `<trim>`:

```
@Builds(rule="<decim>", pop=true, clean=true)
public CountDecimate handleDecimate(Object ... parts)
{
    Integer n = Integer.parseInt((String) parts[0]);
    Processor p = (Processor) parts[1];
    CountDecimate dec = new CountDecimate(n);
    Connector.connect(p, dec);
    add(dec);
    return dec;
}
```

✎

P
"n"
P
n

<decim> := KEEP ONE EVERY <num> FROM (<proc>);

Figure 8.8: A graphical representation of the stack manipulations for rule `<decim>`.

The `<filter>` rule introduces a new element. A `Filter` has two input streams; therefore, one must pop *two* processors from the stack, and connect them in the proper way. This can be done as follows:

```
@Builds(rule="<filter>", pop=true, clean=true)
```

```
public Filter handleFilter(Object ... parts)
{
    Processor p1 = (Processor) parts[0];
    Processor p2 = (Processor) parts[1];
    Filter filter = new Filter();
    Connector.connect(p1, 0, filter, 0);
    Connector.connect(p2, 0, filter, 1);
    add(filter);
    return filter;
}
```

✎

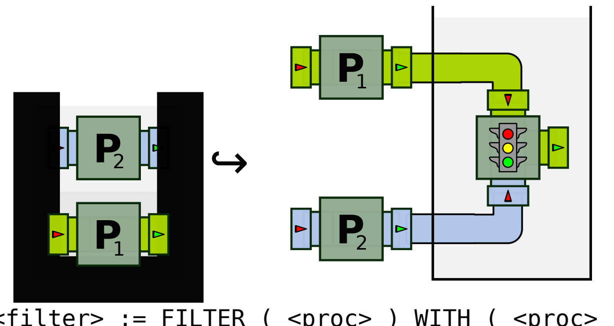


Figure 8.9: A graphical representation of the stack manipulations for rule `<filter>`.

Notice how P1 and P2 are popped from the stack; the output of P1 is connected to the data pipe of a new `Filter` processor, while the output of P2 is connected to its control pipe. Finally, the filter is placed on top of the stack. Remember that objects are popped in the reverse order in which they appear in a rule; however, as per the use of the `pop` annotation, these objects are already popped and given to the method in the correct order by the `GroupProcessorBuilder`. Moreover, because of the `clean` annotation, only the objects corresponding to non-terminal symbols in the grammar rule (underlined) are present in the `parts` array.

The last case in the grammar is that of the `<stream>` rule. According to our grammar, this rule cannot contain another processor expression inside; instead, it is there to designate one of the input pipes at the very beginning of our processor chain. The task of a method handling this rule is therefore to refer to the n-th input of the `GroupProcessor` that is being built. As this rule

is a case of <proc>, it must put a `Processor` on top of the stack.

Internally, the `GroupProcessorBuilder` maintains a set of `Fork` objects for each of the inputs referred to in the query. A call to the `forkInput` method fetches the fork corresponding to the input pipe at position n, adds one new branch to that fork, and connects a `Passthrough` processor at the end of it. This `Passthrough` is then returned. Therefore, the method for <stream> retrieves from the stack a string of digits, converts it into an integer n, and requests a passthrough connected to input pipe n. It then adds this passthrough to the `GroupProcessorBuilder`, puts it on top of the stack, and returns:

```
@Builds(rule="<stream>")
public void handleStream(ArrayDeque<Object> stack)
{
    Integer n = Integer.parseInt((String) stack.pop());
    stack.pop();
    Passthrough p = forkInput(n);
    add(p);
    stack.push(p);
}
```

✎

Graphically, this can be illustrated as follows:

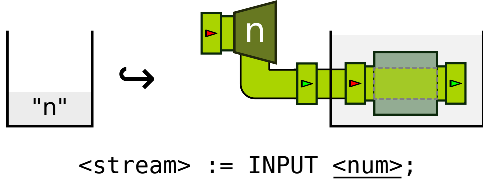


Figure 8.10: A graphical representation of the stack manipulations for rule <stream>.

As we can see on the right-hand side of the figure, a branch of the fork for input n is connected to a `Passthrough` processor and placed on top of the stack.

Done! We have written so far 6 lines of text for the grammar, and less than 40 lines of Java code to implement all the handler methods. The end result is an interpreter that can read expressions in a simple language and produce stream processors from them. Equipped with this builder, we are now ready to parse expressions and use the resulting processors. This works as previously,

with the exception that the output of `build`, this time, is a `Processor` object. Here is an example:

```
Processor proc = builder.build(
    "KEEP ONE EVERY 2 FROM (TRIM 3 FROM (INPUT 0))");
QueueSource src = new QueueSource().setEvents(0, 1, 2, 3, 4, 5, 6, 8);
Connector.connect(src, proc);
Pullable pul1 = proc.getPullableOutput();
for (int i = 0; i < 5; i++)
    System.out.println(pul1.pull());
```

✎

The process is similar to what was done earlier with functions. An instance of the builder is used to parse the expression `KEEP ONE EVERY 2 FROM (TRIM 3 FROM (INPUT 0))`; then, a `QueueSource` is created, and connected to the processor obtained from the builder. From then on, the resulting `Processor` object can be used like any other processor. If the building rules defined earlier were to be applied, step by step, one would discover that the `Processor` returned by `build` is actually this one:

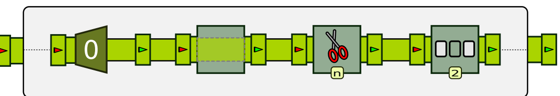


Figure 8.11: The `GroupProcessor` returned by our builder on the expression `KEEP ONE EVERY 2 FROM (TRIM 3 FROM (INPUT 0))`.

The innermost `INPUT 0` corresponds to the `Fork` and the `Passthrough` to the left of the box. The `TRIM 3 FROM` part produces the following `Trim` processor, and the `KEEP ONE EVERY 2 FROM` part produces the `CountDecimate` processor that follows. Finally, the `GroupProcessorBuilder` takes this whole chain and encapsulates it into a `GroupProcessor` of input and output arity 1, connecting input 0 of the box to fork 0, and the output of the chain to output 0 of the box. Note that in this example, since we refer to input pipe 0 only once, the fork and the passthrough are somewhat redundant; further refinements to the `GroupProcessorBuilder` could discover this and connect the input of the group directly to the `Trim` processor *a posteriori*. However, they make the handling of connecting processors to inputs much easier.

In our code example, we pull five events from it and print them to the console; the program displays, unsurprisingly:

```
3
5
8
1
3
```

Mixing Types

Nothing prevents an object builder to create objects of various types. As a more involved example, let us add new rules to the previous builder, which will allow us to create `Function` objects and `ApplyFunction` processors. The grammar could look appear this:

```
<proc>      := <trim> | <decim> | <filter> | <apply> | <stream> ;
<trim>      := TRIM <num> FROM ( <proc> );
<decim>     := KEEP ONE EVERY <num> FROM ( <proc> ) ;
<filter>    := FILTER ( <proc> ) WITH ( <proc> ) ;
<stream>    := INPUT <num> ;
<apply>     := APPLY <fct> ON <proclist> ;
<proclist>  := ( <proc> ) AND ( <proc> ) | ( <proc> ) ;
<fct>       := <add> | <sbt> | <lt> | <abs> | <cons> | <svar> ;
<abs>       := ABS <fct> ;
<add>       := + <fct> <fct> ;
<sbt>       := - <fct> <fct> ;
<lt>        := LT <fct> <fct> ;
<svar>      := X | Y ;
<cons>      := <num> ;
<num>       := ^[0-9]+;
```

A new case has been added to rule `<proc>` to accommodate the `ApplyFunction` processor. The rule `<apply>` has two cases, depending on whether the function given has unary input (requiring a single processor as input), or binary input (in which case the `ApplyFunction` processor must be connected to two inputs). Rules `<fct>` and the following define the syntax to define a function; we reuse the Polish notation from the very first example in the chapter to define functions `<add>`, `<sbt>` and `<abs>` (absolute value). To these functions, `<cons>` and `<svar>` are added, so that `Constant` and `StreamVariable` objects can be used inside function trees.

Stream variables are handled very easily by popping either the string “X” or

“Y”, and by putting the corresponding `StreamVariable` object back onto the stack. This can be done, graphically and in code, as follows:

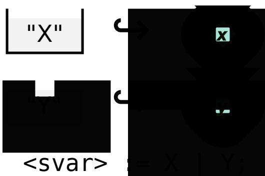


Figure 8.12: A graphical representation of the stack manipulations for rule <svar>.

```
@Builds(rule="<svar>")
public void handleStreamVariable(ArrayDeque<Object> stack)
{
    String var_name = (String) stack.pop();
    if (var_name.compareTo("X") == 0)
        stack.push(StreamVariable.X);
    if (var_name.compareTo("Y") == 0)
        stack.push(StreamVariable.Y);
}
```

✎

Constants work in pretty much the same way. One pops a string from the stack, parses an integer from the string, and then creates a `Constant` object from that integer.

The case for <apply> requires more explanations. This rule first receives a <fct>, corresponding to a `Function` object, which will be encapsulated into an `ApplyFunction` processor. However, depending on whether the function has an input arity of 1 or 2, this processor must be connected to either one or two upstream processors –and hence, either one or two such objects must be popped from the stack. This is the purpose of the <proclist> non-terminal symbol. As one can see, the rule for <proclist> has two cases; the first case corresponds to a construct containing two <proc> expressions, and the second case corresponds to a construct with a single <proc> expression.

The method handling <proclist> is written as follows:

```
@Builds(rule="<proclist>")
public void handleProcList(ArrayDeque<Object> stack)
{
    List<Processor> list = new ArrayList<Processor>();
```

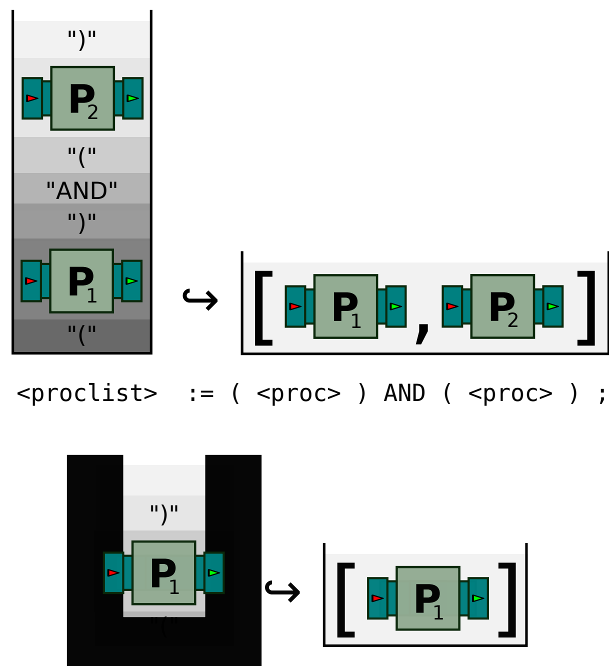
```
    stack.pop();
    list.add((Processor) stack.pop());
    stack.pop();
    if (stack.peek() instanceof String &&
        ((String) stack.peek()).compareTo("AND") == 0)
    {
        stack.pop();
        stack.pop();
        list.add((Processor) stack.pop());
        stack.pop();
    }
    stack.push(list);
}
```

✎

The method first creates an empty `List` of `Processor` objects. It then pops three objects; the second is a `Processor` that is put into the list, and the other two are discarded. This is due to the fact that both cases of rule `<proclist>` end with the same three tokens: `( <proc> )`. The method then *peeks* (but does not pop) the next element on the stack. If this element is the string "AND", we are in the first case of the rule, and four more tokens are popped. This corresponds to the first half of the case, `( <proc> ) AND`. A second processor is extracted from this piece of code and added to the list. The method then pushes back onto the stack the `List` object, which contains either one or two processors. Graphically, this can be represented as in Figure 8.13.

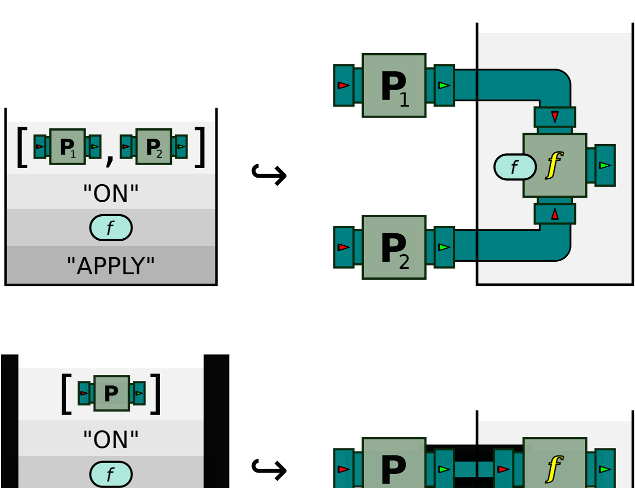
The case for `ApplyFunction` now becomes easy. The method simply pops a `Function` object and a `List` object. Depending on the size of the list, it connects either one or two processors from that list to `ApplyFunction`, and puts it back on the stack.

```
@Builds(rule="<apply>", pop=true, clean=true)
public Processor handleApply(Object ... parts)
{
    Function f = (Function) parts[0];
    ApplyFunction af = new ApplyFunction(f);
    List<Processor> list = (List<Processor>) parts[1];
    if (list.size() == 1)
    {
        Connector.connect(list.get(0), af);
    }
    else if (list.size() == 2)
```



")"
P2
"("
"AND"
")"
P1
"("
P1 , P2
<proclist> := (<proc>) AND (<proc>) ;
")"
P1
P1
<proclist> := (<proc>) ;

↪




```
<proclist>  := APPLY <fct> ON <proclist> ;
```

As one can see, the processor from the stack is connected to the very beginning of the chain, and the very end of the chain is put back onto the stack. This is to show that a grammar construct does not need to instantiate a single `Processor` object. A single grammar rule can result in the creation of multiple objects at once.

Equipped with these new rules, users can write expressions that use the `ApplyFunction` processor and create functions. For example, from an expression such as:

```
APPLY + X Y ON (
  FILTER (INPUT 0)
  WITH (
    APPLY LT X 0 ON (INPUT 0)
))
AND (
  THE AVERAGE OF (INPUT 1))
```

...the object builder will create the following `GroupProcessor`:

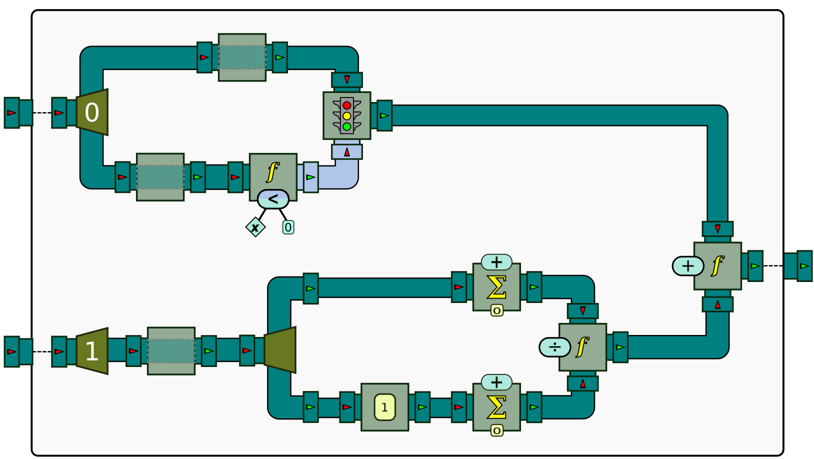


Figure 8.16: The `GroupProcessor` created by a complex query mixing functions and various other types of processors.

The complete object builder for this grammar requires 15 rules and roughly 130 lines of code for the interpreter.

In this chapter, we have seen why BeepBeep does not provide a single built-in

query language to write processor chains. Rather, using a palette called `dsl`, it provides facilities that allow users to design and use their own domain-specific language. The `dsl` palette makes it possible to quickly write the *grammar* for a language, and provides a *parser* called Bullwinkle that can read and parse strings from any grammar at runtime. Moreover, thanks to a special object called a `GrammarObjectBuilder`, one can easily walk through a parsing tree, and progressively construct an object such as a chain of processors by defining methods specific to each rule of the grammar. The end result is that, through a few lines of grammar and a few lines of building code, it is possible to have a working interpreter for a custom query language with very little effort.

Remember that the languages shown in this chapter are only *examples* meant to illustrate the usage of the Bullwinkle parser and its various `ObjectBuilders`. They are no more “BeepBeep’s language” than any language users can create themselves: as we have seen at the beginning of the chapter, this is actually the whole point. The syntax for the languages created does not have to look even remotely like the examples provided. Although this might sound a little tacky, the limit here truly is one’s imagination!

A

Drawing Guide

As you may have noticed, in many cases the best way of understanding a chain of processors is to represent it graphically.

A straightforward way of showing processor chains would be to depict processors as shapes, perhaps with a text label indicating what they do, and to use straight lines and arrows to illustrate their interlinking. Indeed, this is what is often done to represent composition in other systems, such as this example using Apache Storm Trident:

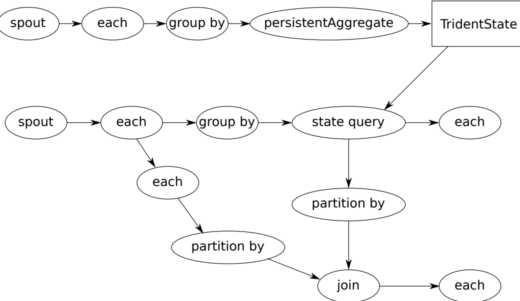


Figure A.1: The composition of "spouts" and "bolts" in Apache Storm Trident.

However, we found early on that BeepBeep processor chains illustrated in such a way are neither particularly intuitive (all processor chains look alike until you start reading what's in the boxes) nor very pleasing to the eye (after all, we have been producing color and graphics on computer monitors for at least thirty years). Therefore, we decided to develop a more colorful, intuitive, yet standardized way of drawing chains of processors.

In this appendix, the basic "rules" defining how to draw pipes, processors and functions are described; these conventions have been followed throughout

this book, in the online documentation, as well as in all BeepBeep presentations given at scientific conferences in current years.

No need to be an artist to create your own processor chains. Using a vector drawing program such as Inkscape (freely available for all operating systems), it is easy to copy-paste the symbols from this book and include them in other drawings. A PDF document containing multiple pages of predefined symbols can also be obtained from the BeepBeep GitHub repository (look for a file called `Drawing Guide`).

Pipes

The **head** of a pipe should indicate whether it is an input or an output pipe. This is done with the inward-pointing *red* triangle (for input) and the outward-pointing *green* triangle (for output).

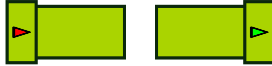


Figure A.2: Input and output pipes.

In longer pipes, only the main body is longer; the head is not stretched.

Figure A.3: A longer pipe.

The **colour** of a pipe should indicate the type of the events it contains. For the sake of consistency, we try to use the same colour for the same type across a diagram. At the very least, frequent event types should have the same colour across a common set of examples. Here are the colours that have been used for frequent event types in this book:

Pipe segments should be either vertical or horizontal. Orientation changes are done through rounded right-angle turns.

Pipe segments can be joined by vertically or horizontally centering them, overlapping them slightly, and using the *Path/Union* command in Inkscape.

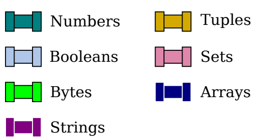


Figure A.4: Colour coding for pipes.

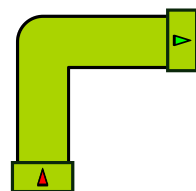


Figure A.5: A pipe with a 90-degree angle.

Processors

Processors are represented by (square) boxes with input/output pipes around them. A symbol in the center of the box represents the processor's specific functionality. The colour of the input/output pipes should match the type of the corresponding input/output stream.

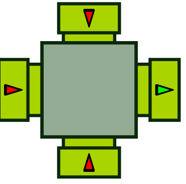


Figure A.6: A generic processor box.

For a processor that takes parameters, these parameters should be placed across one of the unused sides of the processor's box.

Figure A.7: Processors taking parameters.

For processors that have as parameter another processor chain or a function tree, a link to that object is drawn with a circle and a line.

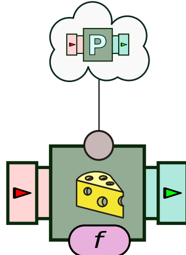


Figure A.8: Processor taking another processor as a parameter.

The “cloud” can be replaced by a rectangle for better legibility.

Functions

Atomic functions are represented by rounded rectangles. When possible, their colour should have a similar shade to that of the input or output type of their arguments.

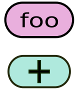


Figure A.9: Functions.

Function *trees* (i.e. composition of multiple atomic functions) are drawn as trees. For functions that need arguments from multiple input streams, the position of the stream is explicitly written in a lozenge. Stream numbers start at 1.

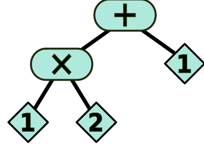


Figure A.10: A function tree.

The edges can be collapsed if the resulting drawing is legible enough.

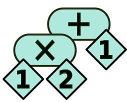


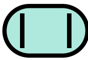
Figure A.11: A collapsed function tree.

B Glossary

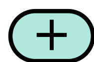
This book has introduced many concepts revolving around the concept of event streams. In particular, the BeepBeep library provides a little “zoo” of dozens of `Processor` and `Function` objects. In this appendix, you will find a list of the various objects and notions that have been discussed. For entries referring to Java objects (such as processors and functions), a note next to the entry indicates whether these objects are part of BeepBeep's core, or can be found in one of its auxiliary palettes.

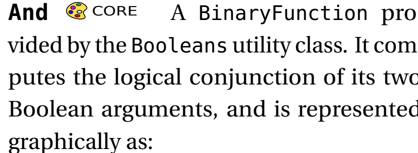
For more technical information about each of these objects, the reader is referred to the online API documentation, which provides in-depth and up-to-date information.

`AbsoluteValue` CORE An `UnaryFunction` provided by the `Numbers` utility class. It computes the absolute value of a number. It is represented as:



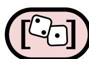
`Addition` CORE A `BinaryFunction` provided by the `Numbers` utility class. It adds two numbers. It is represented as:



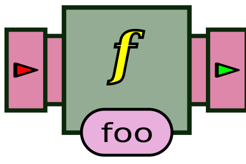


`And` CORE A `BinaryFunction` provided by the `Booleans` utility class. It computes the logical conjunction of its two Boolean arguments, and is represented graphically as:

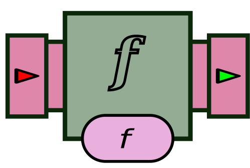
`AnyElement` CORE A 1:1 `Function` provided by the `Bags` utility class. This function takes as input a Java `Collection` *c*, and returns as its output an arbitrary element of *c*. It is represented graphically as:



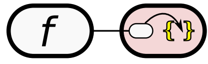
`ApplyFunction` CORE A `Processor` that applies a specific function *f* to every event *front* it receives, and returns the output of that function. The input and output *arity* of this processor are equal to the input and output arity of the underlying function. It is represented graphically as:



`ApplyFunctionPartial` CORE A variant of `ApplyFunction` that attempts to evaluate a function on incomplete input event fronts. It is represented graphically as:



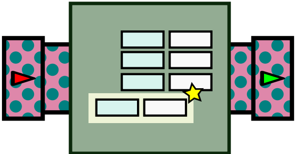
`ApplyToAll` CORE A `1:1 Function` provided by the `Bags` utility class. This function takes as input a Java `Collection` *c* and returns as its output the collection that is the result of applying a predefined `1:1 Function` *f* to each element of *c*. It is represented graphically as:



Arity For a `Processor` object, refers to the number of pipes it has. The *input arity* is the number of input streams accepted by the processor, and the *output arity* is the number of output streams it produces.

For a `Function` object, refers to the number of arguments it accepts or the number of values it produces.

`ArrayPutInto` CORE A `Processor` provided by the `Maps` utility class. Updates a map by putting key-value pairs into it. The processor takes a single input stream, whose events are *arrays* of size 2, and repeatedly outputs a reference to the same internal `Map` object. The first element of the array contains the key, and the second contains the value that will be put into the array. Upon each input array, the processor outputs the map, updated accordingly. The processor is represented graphically as:

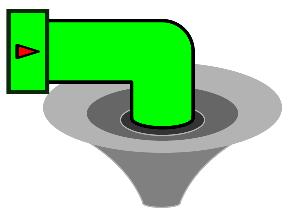


See also `PutInto`.

`Bags` CORE A container class for functions and processors applying to generic collections, i.e. "bags" of objects. Among the processors and functions provided by `Bags` are: `AnyElement`, `ApplyToAll`, `Contains`, `Explode`, `FilterElements`, `GetSize`, `Product`, `RunOn`, `ToArray`, `ToCollection`, `ToList`, and `ToSet`.

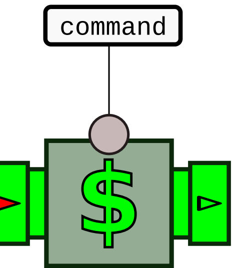
`BinaryFunction` CORE A `Function` object having exactly two input arguments, and producing exactly one output value.

`BlackHole` CORE A special type of `Sink` that discards everything it receives. It is represented graphically as follows:



`Booleans` CORE A container class for `Function` objects related to Boolean values. For example, the static reference `Boolean.and` refers to the `Function` computing the logical conjunction of two Booleans. Among the functions provided by `Booleans` are: `And`, `Implies`, `Not` and `Or`.

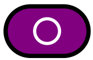
`Call` CORE A `Processor` object calling an external command upon receiving an event, and returning the output of that command as its output stream.



`CallbackSink` CORE A `Sink` that calls a method when a new front of events is pushed to it. Users can override that method to do some processing on these events.

Closed (chain) A property of a chain of processors, when either all its downstream processors are `Sink`s, or all its upstream processors are `Source`s. A chain of processors that is not closed will generally throw Java exceptions when events pass through it.

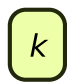
`Concat` CORE A `BinaryFunction` provided by the `Strings` utility class. It receives two strings as arguments, and returns the concatenation of both strings as its output value. It is represented as:



`Connector` CORE A utility class that provides a number of convenient methods for connecting the outputs of processors to the inputs of other processors. Methods provided by the `Connector` class are called `connect()` and have various signatures. When called with exactly two `Processor` arguments, `connect` assigns each output pipe of the first processor to the input pipe at the same position on the second processor.

`Constant` CORE A `Function` object that takes no input argument, and returns a single output value. Constants are used in `FunctionTree`s to refer to fixed values. A `Constant` instance can be created out

of any Java object, and returns this object as its value. When depicted in a `FunctionTree`, they are generally represented as values inside a rounded rectangle, as follows:



`Contains (Bags)` CORE A 2:1 `Function` provided by the `Bags` utility class. This function takes as input a Java `Collection` *c* and an object *o*, and returns the Boolean value `true` as its output if and only if *o* is an element of *c*. It is represented graphically as:

`Contains (Strings)` CORE A `BinaryFunction` provided by the `Strings` utility class. It receives two strings as input, and returns the Boolean value `true` if the first contains the second. It is represented as:

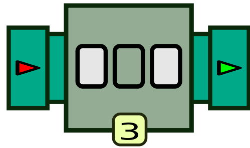
`Context` CORE An associative (key-value) map used by `Processor` objects to store persistent data. Each processor has its own `Context` object. When a processor is cloned, the context of the original is copied into the clone. In addition, all operations on a `Context` object are synchronized.

`ContextAssignment` CORE An object that defines the value associated to a key in a `Processor`'s `Context` object. It is represented graphically as:

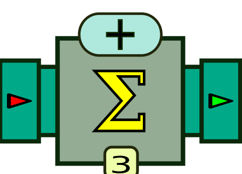
`ContextVariable` CORE A `Function` object that acts as a placeholder for the value associated to a key in a the `Context` of a `Processor`. When a `ContextVariable` occurs inside the `FunctionTree` assigned to an `ApplyFunction` processor, it queries that processor's `Context` object to get the current value associated to the key. It is represented graphically as:

By convention, context variables are prefixed with a dollar sign in diagrams, to differentiate them from constants.

`CountDecimate` CORE A `Processor` that returns every *n*-th input event (starting with the first). The value *n* is called the **decimation interval**. However, a mode can be specified in order to output the *n*-th input event if it is the last event of the trace and it has not been output already. It is represented graphically as:

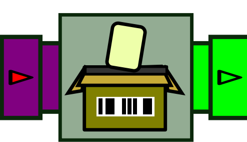


`Cumulate` CORE A `Processor` that creates a cumulative processor out of a cumulative function. This is simply an instance of `ApplyFunction` whose function is of a specific type (a `CumulativeFunction`). It is represented graphically as:



`CumulativeFunction` CORE A special type of `Function` with memory.

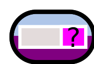
`Deserialize` SERIALIZATION A `Processor` that takes structured character strings as its inputs, and turns each of them into Java objects with the corresponding content. It is represented graphically as follows:



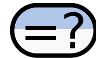
Deserialization can be used to restore the state of objects previously saved on a persistent medium, or to receive non-primitive data types over a communication medium such as a network. The opposite operation is called *serialization*.

`Division` CORE A `BinaryFunction` provided by the `Numbers` utility class. It computes the quotient of two numbers. It is represented as:

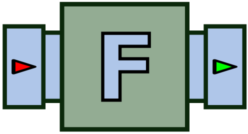
`EndsWith` CORE A `BinaryFunction` provided by the `Strings` utility class. It receives two strings as arguments, and returns `true` if the former ends with the latter.



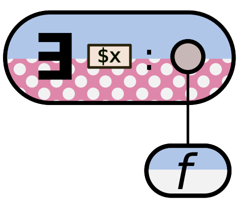
`Equals` CORE A `Function` that checks for the equality between two objects. It is represented graphically as follows:



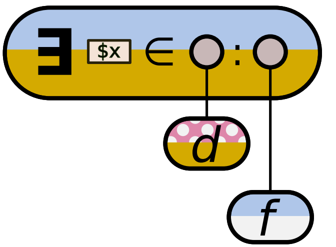
`Eventually` LTL A `Processor` that implements the "eventually" or **F** operator of Linear Temporal Logic. If p is an LTL expression, **F** p stipulates that p should evaluate to `true` on at least one suffix of the current trace. It is represented graphically as:



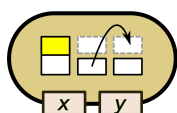
`Exists` FOL A `Function` that acts as an existential quantifier in first-order logic. It is represented as:



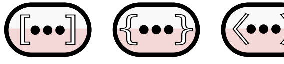
There is also a variant that uses an auxiliary function to compute the set of values to quantify over. It is represented as:



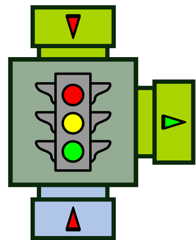
ExpandAsColumns TUPLES A `Function` that transforms a tuple by replacing two key-value pairs by a single new key-value pair. The new pair is created by taking the value of a column as the key, and the value of another column as the value. It is represented as:



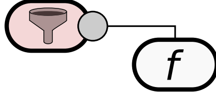
Explode CORE A 1:m `Function` provided by the `Bags` utility class. Given a collection of size m, it returns as its m outputs the elements of the collection. It can be seen as the opposite of `ToArray`, `ToList` and `ToSet`. The ordering of the arguments is ensured when the input collection is itself ordered. The pictogram used to represent the function depends on the type of collection used for the input, in order to match the pictograph of the inverse function.



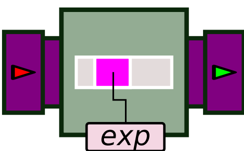
Filter CORE A `Processor` that discards events from an input trace based on a selection criterion. The processor takes as input two events simultaneously; it outputs the first if the second is true. Graphically, this processor is represented as:



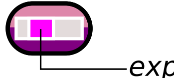
FilterElements CORE A 1:1 `Function` provided by the `Bags` utility class. This function is parameterized by a 1:1 `Function` f that must return a Boolean. The `FilterElements` function takes as input a Java `Collection` c and returns as its output the collection consisting of only the elements of c for which $f(c)$ returns `true`. It is represented graphically as:



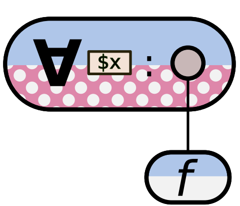
FindPattern A `Processor` that extracts chunks of an input stream based on a regular expression. It is represented graphically as:



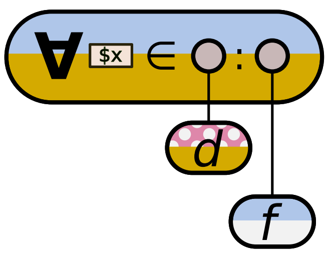
`FindRegex` CORE An `UnaryFunction` provided by the `Strings` utility class. It receives a string *s* as its argument, and returns an *array* of strings, corresponding to all the matches of a given regular expression *exp* on *s*. It is represented as:



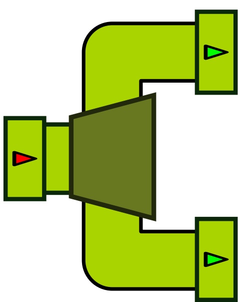
`ForAll` FOL A `Function` that acts as a universal quantifier in first-order logic. It is represented as:



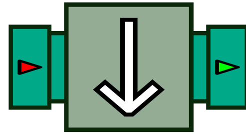
There is a variant of `ForAll` that uses an auxiliary function to compute the set of values to quantify over. It is represented as:



`Fork` CORE A `Processor` that duplicates a single input stream into two or more output streams. A `Fork` is used when the contents of the same stream must be processed by multiple processors in parallel. It is represented graphically as:



`Freeze` CORE A `Processor` that repeatedly outputs the first event it has received. It is represented graphically as:

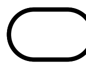


Front Given *n* streams, the front at position *k* is the tuple made of the event at the *k*-th position in every stream. BeepBeep `Processors` that have an input *arity* greater than 1 handle events one front at a time; this is called *synchronous processing*.

`Function` CORE A computation unit that receives one or more *arguments*, and produces one or more output *values*. Along with `Processor`, this is one of BeepBeep's fundamental classes. Contrary to processors, functions are *state-*

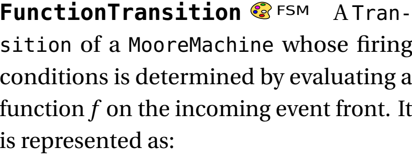
less (or history-independent): the same inputs must always produce the same outputs.

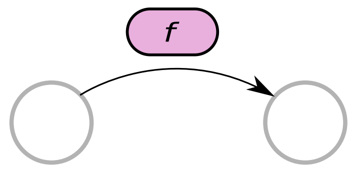
Functions are represented graphically as rounded rectangles, with a pictogram describing the computation they perform, such as this:



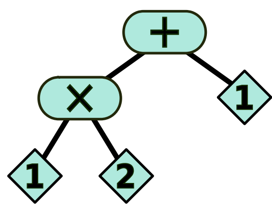
A function with an input arity of m and an output arity of n is often referred to as an m:n function.

`FunctionTransition` FSM A `Transition` of a `MooreMachine` whose firing conditions is determined by evaluating a function f on the incoming event front. It is represented as:





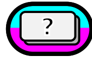
`FunctionTree` CORE A `Function` object representing the composition of multiple functions together to form a "compound" function. A function tree has a *root*, which consists of an m:n function. This function is connected to n children, which can be functions or function trees themselves. The diagram below depicts a function tree that composes multiplication and addition to form a more complex function of two arguments.



`Get` CORE A `UnaryFunction` provided by the `Maps` utility class; it is parameterized by a key k. Given a `Map` m as its input, it retrieves the value associated to k in m. It is represented graphically as:

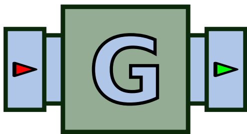
`GetSize` CORE A 1:1 `Function` provided by the `Bags` utility class. This function takes as input a Java `Collection` c and returns as its output the number of elements in that collection. It is represented graphically as:

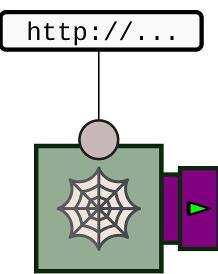
`GetWidgetValue` WIDGETS A `Function` that takes a Swing component as input, and returns the current "value" of this component. It is represented graphically as:



`Globally` LTL A `Processor` that implements the "globally" or **G** operator of Linear Temporal Logic. If p is an LTL expression, **G** p stipulates that p should eval-

uate to `true` on every suffix of the current trace. It is represented graphically as:





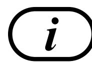
`GroupProcessor` CORE A `Processor` that encapsulates a chain of processors as if it were a single object. It is represented as follows:

To create a `GroupProcessor`, one must first instantiate and connect the processors to be encapsulated. Each processor must then be added to the group through a method called `add`. Finally, the endpoints of the chain must be associated to the inputs and outputs of the group. From then on, the processor can be moved around, connected and duplicated as if it were a single processor.

In a graphical representation of a `GroupProcessor`, the processor chain inside the group can also be drawn.

`HttpGet` CORE A `Source` that reads chunks of data from an URL, using an HTTP request. These chunks are returned as events in the form of strings. It is represented as:

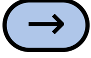
`IdentityFunction` CORE A `Function` that returns its input for its output. It is represented as follows:



The actual colour of the oval depends on the type of events that the function relays.

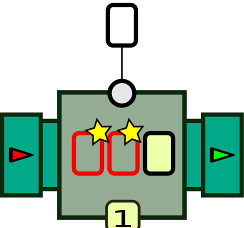
`IfThenElse` CORE A 3:1 `Function` that acts as an if-then-else. If its first input is true, it returns its second input; otherwise it returns its third input. It is represented as follows:

`Implies` CORE A `BinaryFunction` provided by the `Booleans` utility class. It computes the logical implication of its two Boolean arguments, and is represented graphically as:



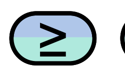
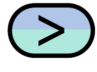
`Insert` CORE A `Processor` that inserts an event a certain number of times before letting the input events through.

This processor can be used to shift events of an input trace forward, by padding the beginning of the trace with some dummy element. It is represented graphically as:

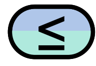
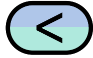


IsEven CORE An `UnaryFunction` provided by the `Numbers` utility class. It returns the Boolean value `true` if and only if its argument is an even number. It is represented as:

IsGreaterOrEqual, IsGreaterThan CORE Two `BinaryFunctions` provided by the `Numbers` utility class. They return the Boolean value `true` if their first argument is greater than (or equal to) the second argument. They are represented as:

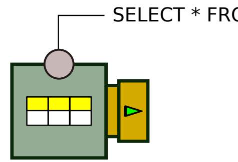



IsLessOrEqual, IsLessThan CORE Two `BinaryFunctions` provided by the `Numbers` utility class. They return the Boolean value `true` if their first argument is less than (or equal to) the second argument. They are represented as:

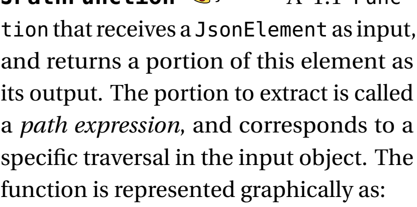



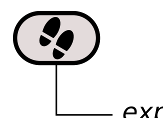
IsSubsetOrEqual CORE A `BinaryFunction` that receives two sets as arguments, and returns `true` if the first is a subset of the second. It is represented as:

JdbcSource JDBC A `Source` object that executes an SQL query on a JDBC connection, and returns the result as a sequence of tuples. It is represented graphically as:



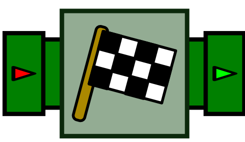
JPathFunction JSON A 1:1 `Function` that receives a `JsonElement` as input, and returns a portion of this element as its output. The portion to extract is called a *path expression*, and corresponds to a specific traversal in the input object. The function is represented graphically as:



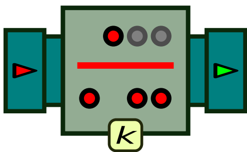


JsonElement JSON An object representing a part of a JSON document. Specific types of JSON elements are `JsonBoolean`, `JsonList`, `JsonMap`, `JsonNull`, `JsonNumber`, and `JsonString`.

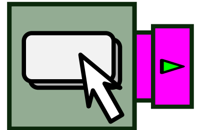
`KeepLast` CORE A `Processor` that returns only the very last event of its input stream, and discards all the previous ones. It is represented graphically as:



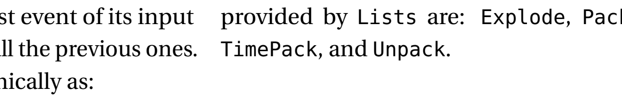
`Limit` SIGNAL A 1:1 `Processor` that receives a stream of numerical values; if the processor receives a non-zero value, it outputs this value, but will turn the *k* following ones into 0, whether they are null or not. Graphically, this processor is represented as:



`ListenerSource` WIDGETS A `Source` processor that wraps around a Swing component, and pushes `ActionEvents` or `ChangeEvents` when user actions are performed on the component. It is represented as a box illustrating the widget that is being wrapped, such as follows:

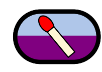


`Lists` CORE A container class for functions and processors applying to ordered collections (`Lists`) and arrays. Among the processors and functions provided by `Lists` are: `Explode`, `Pack`, `TimePack`, and `Unpack`.



`Maps` CORE A container class for functions and processors applying to Java `Maps`, i.e. associative key-value arrays. Among the processors and functions provided by `Maps` are: `ArrayPutInto`, `Get`, `PutInto`, and `Values`.

`Matches` CORE A `BinaryFunction` provided by the `Strings` utility class. It receives two strings as its arguments, and returns the Boolean value `true` if the first matches the regular expression defined in the second. It is represented as:

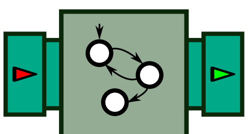


`Maximum` CORE A `BinaryFunction` provided by the `Numbers` utility class. It returns the maximum of its two arguments. It is represented as:

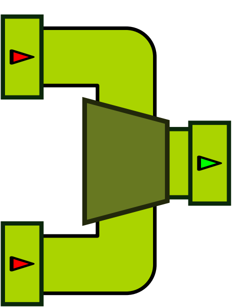
`Minimum` CORE A `BinaryFunction` provided by the `Numbers` utility class. It returns the minimum of its two arguments. It is represented as:

`MooreMachine` FSM A `Processor` that receives an event stream and which,

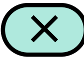
upon each input event, updates its internal state according to a deterministic finite state machine. Each state can be associated with an event to output, corresponding to the formal definition of a Moore machine in theoretical computer science. The `MooreMachine` is depicted by the graph of the FSM it implements; if the graph is too cumbersome, a generic box can be used instead:



`Multiplex` CORE A `Processor` that merges the contents of multiple streams into a single stream. It is an m:1 processor that outputs an event as soon as one is available on one of its input pipes. It is represented graphically as:

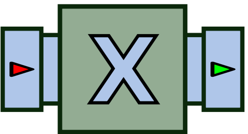


`Multiplication` CORE A `BinaryFunction` provided by the `Numbers` utility class. It computes the product of two numbers. It is represented as:

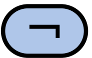


`Multiset` CORE A `Set` that preserves the multiplicity of its elements.

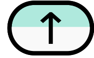
`Next` LTL A `Processor` that implements the "next" or **X** operator of Linear Temporal Logic. If p is an LTL expression, **X** p stipulates that p should evaluate to `true` on the suffix of the current trace starting at the next event. It is represented graphically as:



`Not` CORE An `UnaryFunction` provided by the `Booleans` utility class. It computes the logical negation of its Boolean argument, and is represented graphically as:



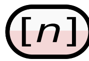
`NumberCast` CORE An `UnaryFunction` provided by the `Numbers` utility class. It attempts to convert an arbitrary Java `Object` into a number. It is represented as:



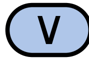
`Numbers` CORE A container class for functions applying to Java `Numbers`. Among the functions provided by `Numbers` are: `AbsoluteValue`, `Addition`, `Division`, `IsEven`, `IsGreaterOrEqual`, `IsGreaterThan`, `IsLessOrEqual`,

IsLessThan, Maximum, Minimum, Multiplication, NumberCast, Power, Signum, SquareRoot, and Subtraction.

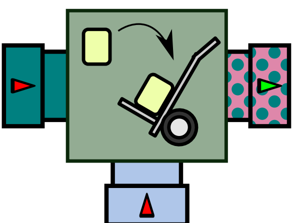
NthElement CORE An UnaryFunction that returns the n-th element of an ordered collection (array or list). It is represented graphically as:



Or CORE A BinaryFunction provided by the Booleans utility class. It computes the logical disjunction of its two Boolean arguments, and is represented graphically as:



Pack CORE A Processor provided by the Lists utility class. It accumulates events from a first input pipe, and sends them in a burst into a list based on the Boolean value received on its second input pipe. A value of true triggers the output of a list, while a value of false accumulates the event into the existing list. This processor is represented graphically as follows:

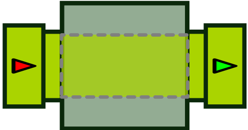


The opposite of Pack in Unpack. See also TimePack.

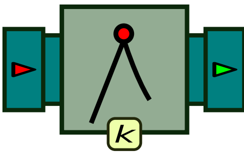
ParseJson JSON A Function that turns a character string into a structured object called a JsonElement. The function is represented graphically as:

ParseXml XML A Function that turns a character string into a structured object called an XmlElement. The function is represented graphically as:

Passthrough CORE A Processor object that lets every input event through as its output. This processor can be used as a placeholder when a piece of code needs to be passed a Processor object, but that in some cases, no processing on the events is necessary. Graphically, this processor is represented as:

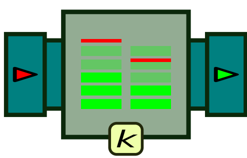


PeakFinder SIGNAL A 1:1 Processor object receives as input a stream of numerical values, and identifies the "peaks" (sudden increases) in that signal. It outputs the value 0 if no peak is detected at the current input position, and otherwise, the height of the detected peak. Graphically, this processor is represented as:

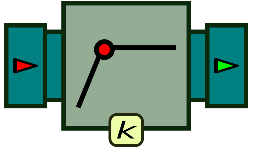


The *Signal* palette implements two variants of `PeakFinder` called `PeakFinderLocalMaximum` and `PeakFinderTravelRise`.

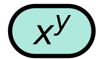
`Persist` SIGNAL A 1:1 `Processor` that receives a stream of numerical values; when a processor receives a non-zero value, it outputs it for the next k events, unless a subsequent input value is greater (in which case this new value is output for the next k events). Graphically, this processor is represented as:



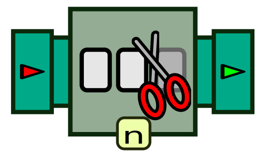
`PlateauFinder` SIGNAL A 1:1 `Processor` object receives as input a stream of numerical values, and identifies the "plateaus" (successive events with similar values) in that signal. It outputs the value 0 if no plateau is detected at the current input position, and otherwise, the height of the detected plateau. Graphically, this processor is represented as:



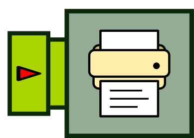
`Power` CORE A `BinaryFunction` provided by the `Numbers` utility class. It computes the first argument, elevated to the power of the second. It is represented as:



`Prefix` CORE A `Processor` that returns the first n input events and discards the following ones. It is represented graphically as:

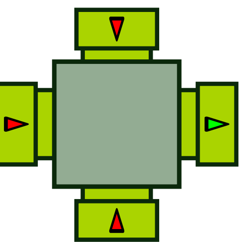


`Print` CORE A `Processor` that sends its input events to a Java `PrintStream` (such as the standard output). This processor takes whatever event it receives (i.e. any Java `Object`), calls its {@link Object#toString() toString()} method, and pushes the resulting output to a print stream. Graphically, it is represented as:



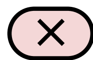
`Processor` CORE A processing unit that receives zero or more input streams, and produces zero or more output streams. The `Processor` is the fundamental class where all stream computation occurs. All of BeepBeep's processors are descendants of this class. A processor is depicted graphically as a "box", with

"pipes" representing its input and output streams.



This class itself is abstract; nevertheless, it provides important methods for handling input/output event queues, connecting processors together, etc. However, if you write your own processor, you will most likely want to inherit from its child, `SynchronousProcessor`, which does some additional work. The `Processor` class does not assume anything about the type of events being input or output. All its input and output queues are therefore declared as containing instances of `Object`, Java's most generic type.

`Product` CORE A 2:1 `Function` provided by the `Bags` utility class. This function takes as input two Java `Collection`, *c1* and *c2*, and returns as its output a set of arrays of size 2, corresponding to the Cartesian product of *c1* and *c2*. It is represented graphically as:

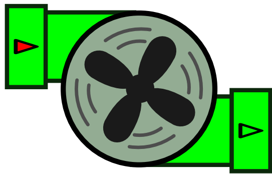


`Pullable` CORE An object that queries events on one of a processor's outputs. For a processor with an output arity *n*, there exists *n* distinct pullables, namely one for each output trace. Every pullable works roughly like a classical `Iterator`: it is possible to check whether new output events are available, and get one new output event. However, contrarily to iterators, `Pullables` have two versions of each method: a *soft* and a *hard* version. The opposite of `Pullables` are `Pushables` –objects that allow users to feed input events to processors. Graphically, a `Pullable` is represented by a pipe connected to a processor, with an outward pointing triangle:

Pull mode One of the two operating modes of a chain of processors. In pull mode, a user or an application obtains references to the `Pullable` objects of the downstream processors of the chain, and calls their `pull()` method to ask for new output events. When using a chain of processors in pull mode, the chain must be closed at its inputs. The opposite mode is called *push mode*.

`Pump` CORE A `Processor` that repeatedly pulls its input, and pushes the resulting events to its output. The `Pump` is a way to bridge an upstream part of a processor chain that works in *pull* mode, to a downstream part that operates in *push* mode.

Graphically, this processor is represented as:



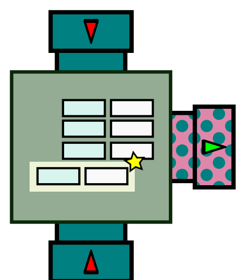
The repeated pulling of events from its input is started by calling this processor's `#start()` method. In the background, this will instantiate a new thread, which will endlessly call pull() on whatever input is connected to the pump, and then call push() on whatever input is connected to it.

The opposite of the `Pump` is the `Tank`.

`Pushable` CORE An object that gives events to some of a processor's input. Interface `Pushable` is the opposite of `Pullable`: rather than querying events form a processor's output (i.e. "pulling"), it gives events to a processor's input. This has for effect of triggering the processor's computation and "pushing" results (if any) to the processor's output. If a processor is of input arity *n*, there exist *n* distinct `Pullables`: one for each input pipe. Graphically, a `Pushable` is represented by a pipe connected to a processor, with an inward pointing triangle:

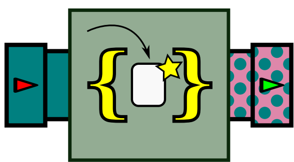
Push mode One of the two operating modes of a chain of processors. In push mode, a user or an application obtains references to the `Pushable` objects of the upstream processors of the chain, and calls their `push()` method to feed new input events. When using a chain of processors in push mode, the chain must be closed at its outputs. The opposite mode is called *pull mode*.

`PutInto` (`Maps`) CORE A `Processor` provided by the `Maps` utility class. It updates a map by putting key-value pairs into it. The processor takes two input streams; the first contains the key, and the second contains the value that will be put into the array. Upon each input front, it repeatedly outputs a reference to the same internal `Map` object, updated accordingly. The processor is represented graphically as:



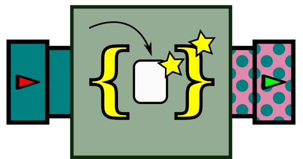
See also `ArrayPutInto`.

`PutInto` (`Sets`) CORE A `Processor` provided by the `Sets` utility class. Updates a set by putting the elements it receives into it. Upon each input event, it repeatedly outputs a reference to the same internal `Set` object, updated accordingly. The processor is represented graphically as:



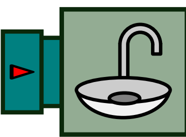
See also `PutIntoNew`.

`PutIntoNew (Sets)` CORE A `Processor` provided by the `Sets` utility class. Updates a set by putting the elements it receives into it. Upon each input event, it creates a new instance of `Set` and adds to it all the events received so far; it then outputs a reference to this new set. The processor is represented graphically as:

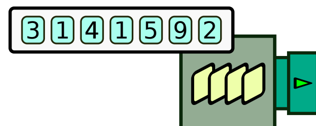


See also `PutInto (Sets)`.

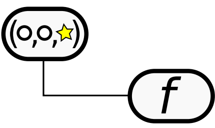
`QueueSink` CORE A `Sink` that accumulates events into queues, one for each input pipe. It is represented graphically as:



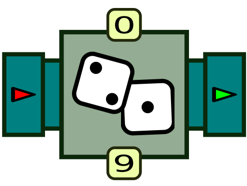
`QueueSource` CORE A `Source` whose input is a queue of objects. One gives the `QueueSource` a list of events, and that source sends these events as its input one by one. When reaching the end of the list, the source returns to the beginning and keeps feeding events from the list endlessly. The `QueueSource` is represented graphically as:



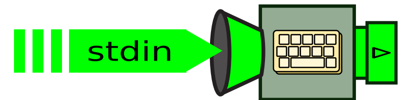
`RaiseArity` CORE A `Function` that raises the arity of another function. Given an m:n function f, an instance of `RaiseArity` r makes f behave like an m':n function, with $m' > m$. The extra arguments given to r are simply ignored. It is represented as:



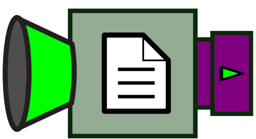
`Randomize` CORE A n:n `Processor` that turns an arbitrary input event front into an output front made of randomly selected numerical values. The interval in which values are selected can be specified. It is represented graphically as:



`ReadInputStream` CORE A `Source` that reads chunks of bytes from a Java `InputStream`. It is represented graphically as follows:

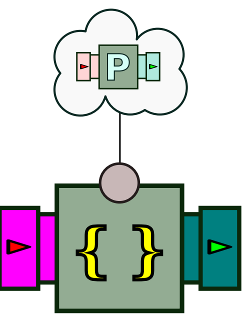


ReadLines CORE A Source that reads entire text lines from a Java `InputStream`. It is represented graphically as:



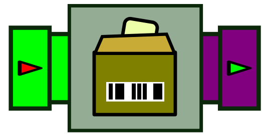
ReadStringStream CORE A variant of `ReadInputStream` that converts the byte chunks into character strings.

RunOn CORE A `Processor` provided by the `Bags` utility class. This processor is parameterized by another processor *P*. It receives as input a stream of collections. On each individual collection *c*, it resets *P*, feeds each element of *c* on *P*, and retrieves the last event output by *P* on that stream; this is the event is the event output by `RunOn` on *c*. It is represented graphically as:



If the collection *c* is unordered and *P* is sensitive to event ordering, the output of `RunOn` on this collection may not always be the same.

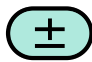
Serialize SERIALIZATION A `Processor` that takes arbitrary objects as its inputs, and turns each of them into a structured character string depicting their content. It is represented graphically as follows:



Serialization can be used to store the state of objects on a persistent medium, or to transmit non-primitive data types over a communication medium such as a network. The opposite operation is called *deserialization*.

Sets CORE A container class for functions and processors applying to Java `Sets` and their descendents. Among the processors and functions provided by `Sets` are: `IsSubsetOrEqual`, `PutInto`, `PutIntoNew`, and `SetUpdateProcessor`.

Signum CORE An `UnaryFunction` provided by the `Numbers` utility class. It returns -1 if the argument is negative, +1 if it is positive, and 0 if the argument is the number 0. It is represented as:



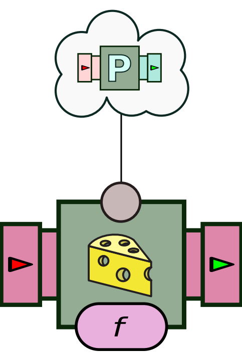
SynchronousProcessor CORE A `Processor` that performs a computation

on input events to produce output events. This is the direct descendant of `Processor`, and probably the one you'll want to inherit from when creating your own processors. While `Processor` takes care of input and output queues, `SynchronousProcessor` also implements `Pullables` and `Pushables`. These take care of collecting input events, waiting until one new event is received from all input traces before triggering the computation, pulling and buffering events from all outputs when either of the `Pullables` is being called, etc. The only thing that is left undefined is what to do when new input events have been received from all input traces. This is the task of abstract method `compute()`, which descendants of this class must implement.

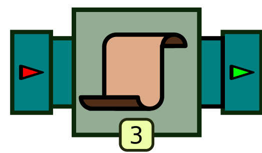
`Sink` CORE A `Processor` with an output *arity* of zero. It is used to close processor chains in *push mode*.

`SinkLast` CORE A variant of `QueueSink` with a queue of size 1. A `SinkLast` remembers only the last event sent to it.

`Slice` CORE A `Processor` that separates the events from an input stream into multiple "sub-streams". A function *f*, called the *slicing function*, dispatches to a copy of P an input event e according to the value of $f(e)$ (there is one copy of P for each possible output value of f). The `Slice` processor returns a Java `Map` containing as keys the value of $f(e)$, and as value, the last event returned by the processor P associated to $f(e)$. It is illustrated as:

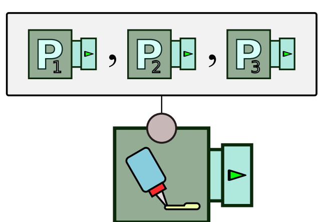


`Smooth` SIGNAL A `Processor` that smoothes a stream of numbers by replacing a value by an average over a window of events. It is illustrated as:



`Source` CORE A `Processor` with an input *arity* of zero. It is used to close processor chains in *pull mode*.

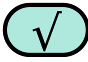
`Splice` CORE A `Source` that joins multiple sources as a single one. The splice processor is given multiple sources. It pulls events from the first one until it does not yield any new event. It then starts pulling events from the second one, and so on. It is illustrated as:



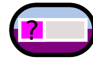
SplitString CORE A `UnaryFunction` provided by the `Strings` utility class. It receives as string as its input, and returns an *array* of strings, split according to a given character separator. It is represented as:

The comma in the figure is to be replaced by the actual character used to separate the input string.

SquareRoot CORE An `UnaryFunction` provided by the `Numbers` utility class. It computes the square root of its argument. It is represented as:



StartsWith CORE A `BinaryFunction` provided by the `Strings` utility class. It receives two strings as its arguments, and returns the Boolean value `true` if the first string starts with the second. It is represented as:

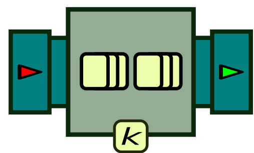


StreamVariable CORE A `Function` standing for the *i*-th stream given as input to a processor. A `StreamVariable` can be given as an argument to a `FunctionTree`. It is represented as follows:

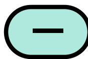
The number inside the diamond represents the stream number. By convention, stream numbers start at 1 in diagrams.

Strings CORE A container class for functions and processors applying to Java `Strings`. Among the processors and functions provided by `Sets` are: `Concat`, `Contains`, `EndsWith`, `FindRegex`, `Matches`, `SplitString`, `StartsWith` and `ToString`.

Stutter CORE A `Processor` that repeats each input event in its output a fixed number of times (*k*; see also `VariableStutter`). It is represented graphically as:



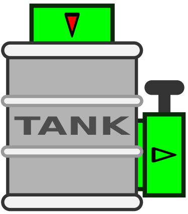
Subtraction CORE A `BinaryFunction` provided by the `Numbers` utility class. It computes the difference of two numbers. It is represented as:



Tank CORE A `Processor` that accumulates pushed events into a queue until they are pulled. The Tank is a way to

bridge an upstream part of a processor chain that works in *push* mode, to a downstream part that operates in *pull* mode.

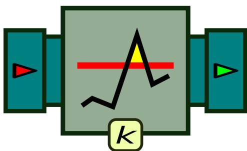
Graphically, this processor is represented as:



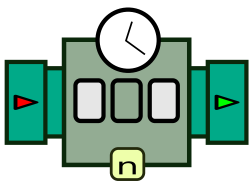
The opposite of the tank is the `Pump`.

`TankLast` CORE A variant of `Tank` which, when pulled, creates an output event based on the last event received.

`Threshold` SIGNAL A 1:1 `Processor` that receives a stream of numerical values; the processor outputs an event if its value is above some threshold value k; otherwise it replaces it by 0. Graphically, this processor is represented as:

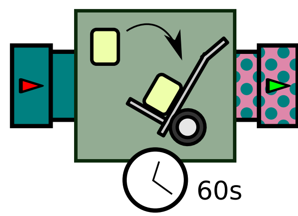


`TimeDecimate` CORE A `Processor` which, after returning an input event, discards all others for the next n seconds. This processor therefore acts as a rate limiter. It is represented as:



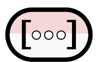
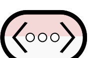
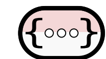
Note that this processor uses `System.currentTimeMillis()` as its clock. Moreover, a mode can be specified in order to output the last input event of the trace if it has not been output already.

`TimePack` CORE A `Processor` provided by the `Lists` utility class. It accumulates events from a first input pipe, and sends them in a burst into a list at predefined time intervals. This processor is represented graphically as follows:



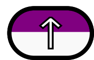
The opposite of `TimePack` in `Unpack`. See also `Pack`.

`ToArray`, `ToList`, `ToSet` CORE Three m:1 `Functions` provided by the `Bags` utility class. Their input arity is defined by parameter m. They turn their m arguments into a Java array, list or set of size m. In the case of arrays and lists, the ordering of the arguments is preserved: the i-th argument of the function is placed at the i-th position in the output collection. The following picture shows the graphical representation of each of these functions:

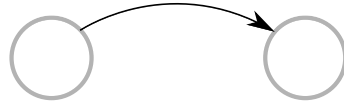
  

ToImageIcon WIDGETS An UnaryFunction that converts an array of bytes containing an image, into a Swing ImageIcon. It is represented as:

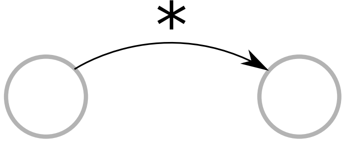
ToString CORE An UnaryFunction provided by the Strings utility class. It attempts to convert an arbitrary Java Object into a String; this is done by calling the object's toString method. It is represented as:



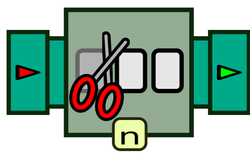
Transition FSM An object used by the MooreMachine processor that indicates how the machine can move between its states. It is represented as:



TransitionOtherwise FSM A Transition object used by the MooreMachine processor that fires only if none of the other outgoing transitions from the same source state fires first.

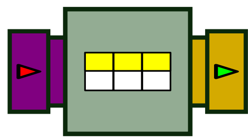


Trim CORE A Processor that discards the first *n* events of its input stream, and outputs the remaining ones as is. It is represented as:

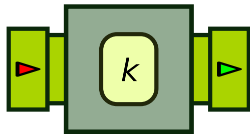


Tuple TUPLES A special type of event defined by BeepBeep's *Tuple* palette, which consists of an associative map between keys and values. Contrary to tuples in relational databases, where values must be scalar (i.e. strings or numbers), the tuples in BeepBeep can have arbitrary Java objects as values (including other tuples).

TupleFeeder TUPLES A Processor that converts lines of text into Tuples. It is represented as:



TurnInto CORE A Processor that turns any input event into a predefined object. It is represented graphically as:

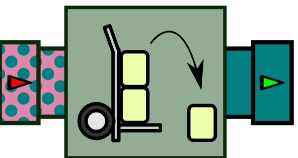


UnaryFunction CORE A Function object that has an input and output *arity*

of exactly 1.

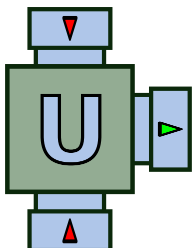
Uniform (processor) A `Processor` that produces the same number of output fronts for every input front it receives. Occasionally, the number of output fronts produced is explicitly mentioned: a k-uniform processor produces exactly k output fronts for every input front.

`Unpack` CORE A `Processor` provided by the `Lists` utility class. It unpacks a list of objects by outputting its contents as separate events. This processor is represented graphically as follows:

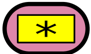


The opposite of `Unpack` is `Pack`.

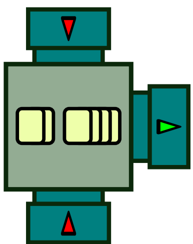
`Until` LTL A `Processor` that implements the "until" or **U** operator of Linear Temporal Logic. If p and q are two streams of Boolean value, p **U** q stipulates that q should evaluate to `true` on some future input front, and that until then, p should evaluates to `true` on every input front. It is represented graphically as:



`Values` CORE A `UnaryFunction` provided by the `Maps` utility class. Given a `Map` m as its input, it returns a `Set` made of all the values present in m. It is represented graphically as:



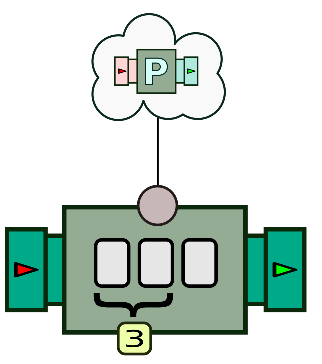
`VariableStutter` CORE A 2:1 `Processor` that repeats each input event coming in its first input pipe a number of times defined by the input event coming into its second input pipe. It is represented graphically as:



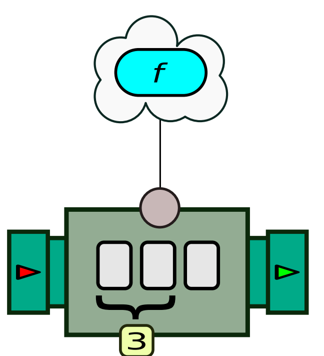
`Variant` A special class that can be returned by a call to a processor's `getInputTypesFor` or `getOutputType` methods. The occurrence of such a type in an input or output pipe disables the type checking step that the `Connector` class normally performs before connecting two processors together.

`Window` CORE A `Processor` that applies another processor on a "sliding window" of events. It takes as arguments another processor P and a window width n. It returns the result of P after processing events 0 to n-1... - Then the result of (a new instance of P) that processes events

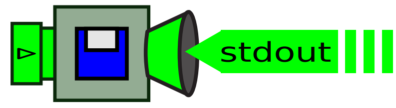
1 to n, and so on. It is represented graphically as:



`WindowFunction` CORE A `Processor` that applies a function on a “sliding window” of events. It takes a sliding window of n successive input events, passes them to the n-ary function f and outputs the result. It is represented graphically as:

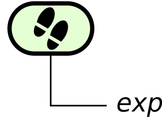


`WriteOutputStream` CORE A `Sink` that writes chunks of bytes to a Java `OutputStream`. It is represented graphically as follows:



`XmlElement` XML An object representing an element of an XML document.

`XPathFunction` XML A 1:1 `Function` that receives an `XmlElement` as input, and returns a portion of this element as its output. The portion to extract is called a *path expression*, and corresponds to a specific traversal in the input object. The function is represented graphically as:



C Further Reading

BeepBeep Research Papers

BeepBeep has been the subject of multiple scientific research papers in the past couple of years. Here is a list of these publications. Note that each of them covers only a specific part of the system and that, depending on its age, may not faithfully reflect the current state of the implementation. Moreover, a few publications (mostly from 2014 and earlier) refer to version 1.x of BeepBeep, which worked differently from the current software. Some of these articles, however, detail interesting use cases where BeepBeep has been involved.

- Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, Sylvain Hallé. (2017). Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring. Computers in Entertainment 15(1): 1:1-1:28, ACM. DOI: 10.1145/2700529.
- Mohamed Recem Boussaha, Raphaël Khoury, Sylvain Hallé. (2017). Monitoring of Security Properties Using BeepBeep. FPS 2017: 160-169. DOI: 10.1007/978-3-319-75650-9_11
- Sylvain Hallé, Raphaël Khoury, Sébastien Gaboury. (2017). Event Stream Processing with Multiple Threads. RV 2017: 359-369. DOI: https://doi.org/10.1007/978-3-319-67531-2_22
- Sylvain Hallé. (2017). From Complex Event Processing to Simple Event Processing. CoRR abs/1702.08051 (2017).
- Sylvain Hallé, Sébastien Gaboury, Bruno Bouchard. (2016). Activity Recognition Through Complex Event Processing: First Findings. AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments 2016. `http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561`
- Sylvain Hallé, Sébastien Gaboury, Raphaël Khoury. (2016). A glue lan-

guage for event stream processing. BigData 2016: 2384-2391. DOI: 10.1109/BigData.2016.7840873

- Raphaël Khoury, Sylvain Hallé, Omar Waldmann. (2016). Execution Trace Analysis Using LTL-FO+. ISoLA (2) 2016: 356-362. DOI: 10.1007/978-3-319-47169-3_26
- Sylvain Hallé, Sébastien Gaboury, Bruno Bouchard. (2016). Towards User Activity Recognition Through Energy Usage Analysis And Complex Event Processing. PETRA 2016: 3. DOI: 10.1145/2910674.2910707
- Sylvain Hallé. (2016). When RV Meets CEP. RV 2016: 68-91. DOI: https://doi.org/10.1007/978-3-319-46982-9_6
- Sylvain Hallé. (2015). A Declarative Language Interpreter for CEP. EDOC Workshops 2015: 156-159. DOI: 10.1109/EDOCW.2015.19
- Sylvain Hallé, Simon Varvaressos. (2014). A Formalization of Complex Event Stream Processing. EDOC 2014: 2-11. DOI: 10.1109/EDOC.2014.12
- Simon Varvaressos, Kim Lavoie, Alexandre Blondin Massé, Sébastien Gaboury, Sylvain Hallé. Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring. ICST 2014: 143-152. DOI: https://doi.org/10.1109/ICST.2014.27
- Simon Varvaressos, Dominic Vaillancourt, Sébastien Gaboury, Alexandre Blondin Massé, Sylvain Hallé. (2013). Runtime Monitoring of Temporal Logic Properties in a Platform Game. RV 2013: 346-351. DOI: 10.1007/978-3-642-40787-1_23
- Sylvain Hallé, Roger Villemaire. (2012). Runtime Enforcement of Web Service Message Contracts with Data. IEEE Trans. Services Computing 5(2): 192-206 (2012). DOI: 10.1109/TSC.2011.10
- Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf, Roger Villemaire. (2010). Runtime Verification of Web Service Interface Contracts. IEEE Computer 43(3): 59-66. DOI: 10.1109/MC.2010.76
- Sylvain Hallé, Roger Villemaire. (2009). Browser-Based Enforcement of Interface Contracts in Web Applications with BeepBeep. CAV 2009: 648-653. DOI: 10.1007/978-3-642-02658-4_50
- Sylvain Hallé, Roger Villemaire. (2008). Runtime Monitoring of Message-Based Workflows with Data. EDOC 2008: 63-72. DOI: 10.1109/EDOC.2008.32

Complex Event Processing Systems

As was mentioned in the introduction, BeepBeep is very close to the field of complex event processing (CEP). Here are a few pointers to books and papers on this topic.

GENERAL LITERATURE

- Neha Narkhede. (2017). Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly. ISBN: 978-1491936160
- Jay Kreps. (2014). I Heart Logs: Event Data, Stream Processing, and Data Integration. O'Reilly. ISBN: 978-1491909386
- David C. Luckham. (2005). The power of events – An introduction to complex event processing in distributed enterprise systems. ACM. ISBN: 978-0-201-72789-0

RESEARCH PAPERS AND TECHNICAL PAPERS

Multiple CEP systems have been mentioned in the introduction. Here are links to research papers and whitepapers about some of these systems.

- Ugur Çetintemel, Daniel J. Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Mike Stonebraker, Nesime Tatbul, Ying Xing, Stan Zdonik. (2016). The Aurora and Borealis Stream Processing Engines. Data Stream Management 2016: 337-359. DOI: 10.1007/978-3-540-28608-0_17
- Remco M. Dijkman, Sander P.F. Peters, Arthur M.F. ter Hofstede. (2016). A Toolkit for Streaming Process Data Analysis. EDOCW 2016: 304-312. DOI: 10.1109/EDOCW.2016.7584341
- Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica. (2016). Apache Spark: a unified engine for big data processing. Communications of the ACM 59(11): 56-65. DOI: 10.1145/2934664
- Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali

Ghodsi, Matei Zaharia. (2015). Spark SQL: Relational Data Processing in Spark. DOI: 10.1145/2723372.2742797

- Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann{-}Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, Daniel Warneke. (2014). The Stratosphere platform for big data analytics. VLDB Journal 23(6): 939-964. DOI: 10.1007/s00778-014-0357-y
- Jian Cao and Xing Wei and Yaqi Liu and Dianhui Mao and Qiang Cai. (2014). LogCEP: – Complex Event Processing based on Pushdown Automaton. Int. Journal of Hybrid Information Technology 7(6): 71-82. DOI: 10.14257/ijhit.2014.7.6.06
- Gianpaolo Cugola, Alessandro Margara. (2012). Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44(3): 15:1-15:62. DOI: 10.1145/2187671.2187677
- Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, Vishaka Nanayakkara. (2011). Siddhi: a second look at complex event processing architectures. SC-GCE 2011: 43-50. DOI: 10.1145/2110486.2110493
- Gerald G. Koch, Boris Koldehofe, Kurt Rothermel. (2010). Cordies: expressive event correlation in distributed systems. DEBS 2010: 10.1145/1827418.1827424
- Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari. (2010). S4: Distributed Stream Computing Platform. ICDMW 2010: 170-177. DOI: 10.1109/ICDMW.2010.172
- Lars Brenna, Johannes Gehrke, Mingsheng Hong, Dag Johansen. (2009). Distributed event stream processing with non-deterministic finite automata. DEBS 2009. DOI: 10.1145/1619258.1619263
- Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, Walker M. White. (2007). Cayuga: a high-performance event processing engine. SIGMOD Conference 2007: 1100-1102. DOI: 10.1145/1247480.1247620
- Eugene Wu, Yanlei Diao, Shariq Rizvi. (2006). High-performance complex event processing over streams. SIGMOD 2006: 407-418. DOI: 10.1145/1142473.1142520
- Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mauyr Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, Jennifer Widom. (2004). STREAM: The Stanford Data Stream Management System. Tech-

nical Report, Stanford InfoLab.
`http://ilpubs.stanford.edu:8090/641/`

- Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, Mehul A. Shah. (2003). TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003.
  `http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf`
- Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, Stanley B. Zdonik. (2002). Monitoring Streams - A New Class of Data Management Applications. VLDB 2002: 215-226.
  `http://www.vldb.org/conf/2002/S07P02.pdf`
- Shivnath Babu, Jennifer Widom. (2001). Continuous Queries over Data Streams. SIGMOD Record 30(3): 109-120. DOI: 10.1145/603867.603884.
- Praveen Seshadri, Miron Livny, Raghu Ramakrishnan. (1994). Sequence Query Processing. SIGMOD 1994: 430-441. DOI: 10.1145/191839.191926
- Nicolas Halbwachs, Fabienne Lagnier, Christophe Ratel. (1992). Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. IEEE Trans. Software Eng. 18(9): 785-793. DOI: 10.1109/32.159839

Runtime Verification and Log Analysis

- Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, Alexander Schramm: TeSSLa: runtime verification of non-synchronized real-time streams. SAC 2018: 1925-1933. DOI: 10.1145/3167132.3167338
- Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, Daniel Thoma. (2016). Runtime Monitoring with Union-Find Structures. ETAPS 2016: 868-884. DOI: 10.1007/978-3-662-49674-9_54
- Martin Leucker. (2016). Runtime Verification for Linear-Time Temporal Logic. SETSS 2016: 151-194. DOI: 10.1007/978-3-319-56841-6_5
- Ariane Piel, Jean Bourrely, Stéphanie Lala, Sylvain Bertrand, Romain Kervarc. (2016). Temporal Logic Framework for Performance Analysis of Architectures of Systems. NFM 2016: 3-18. DOI: 10.1007/978-3-319-40648-0_1

- David A. Basin, Felix Klaedtke, Srdjan Marinovic, Eugen Zalinescu. (2015). Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design. 45 (3): 262-285. DOI: 10.1007/s10703-015-0222-7.
- Yliès Falcone, Klaus Havelund, Giles Reger. A Tutorial on Runtime Verification. (2013). Engineering Dependable Software Systems 2013: 141-175. DOI: 10.3233/978-1-61499-207-3-141
- Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, David E. Rydeheard. (2012). Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. FM 2012: 68-84. DOI: 10.1007/978-3-642-32759-9_9
- Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, Grigore Rosu (2012). An overview of the {MOP} runtime verification framework. STTT 14(3): 249-289. DOI: 10.1007/s10009-011-0198-6
- Howard Barringer, Klaus Havelund. (2011). TraceContract: A Scala DSL for Trace Analysis. FM 2011: 57-72. DOI: 10.1007/978-3-642-21437-0_7
- Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, Wil M. P. van der Aalst. (2011). Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. BPM 2011: 132-147. DOI: 10.1007/978-3-642-23059-2_13
- Howard Barringer, Alex Groce, Klaus Havelund, Margaret H. Smith. (2010). Formal Analysis of Log Files. JACIC 7(11): 365-390. DOI: 10.2514/1.49356
- David A. Basin, Felix Klaedtke, Samuel Müller. (2010). Policy Monitoring in First-Order Temporal Logic. CAV 2010: 1-18. DOI: 10.1007/978-3-642-14295-6_1
- Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, Nomair A. Naeem. (2010). Collaborative Runtime Verification with Tracematches. J. Log. Comput. 20(3): 707-723. DOI: 10.1093/logcom/exn077
- Christian Colombo, Andrew Gauci, Gordon J. Pace. (2010). LarvaStat: Monitoring of Statistical Properties. RV 2010: 480-484. DOI: 10.1007/978-3-642-16612-9_38
- Kari Kähkönen, Jani Lampinen, Keijo Heljanko, Ilkka Niemelä. (2009). The LIME Interface Specification Language and Runtime Monitoring Tool. RV 2009: 93-100. DOI: 10.1007/978-3-642-04694-0_7
- Feng Chen, Grigore Rosu. (2009). Parametric Trace Slicing and Monitoring. TACAS 2009: 246-261. DOI: 10.1007/978-3-642-00768-2_23
- Martin Leucker, Christian Schallhart. (2009). A brief account of runtime verification. J. Log. Algebr. Program. 78 (5): 293–303. DOI:

10.1016/j.jlap.2008.08.004

- Ahmed Awad, Gero Decker, Mathias Weske. (2008). Efficient Compliance Checking Using BPMN-Q and Temporal Logic. BPM 2008: 326-341. DOI: 10.1007/978-3-540-85758-7_24
- Federico Chesani, Paola Mello, Marco Montali, Fabrizio Riguzzi, Maurizio Sebastianis, Sergio Storari. (2008). Checking Compliance of Execution Traces to Business Rules. BPM-W 2008: 134-145. DOI: 10.1007/978-3-642-00328-8_13
- Christian Colombo, Gordon J. Pace, Gerardo Schneider. (2008). Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. FMICS 2008: 135-149. DOI: 10.1007/978-3-642-03240-0_13
- Aditya Ghose and George Koliadis. (2007). Auditing Business Process Compliance. ICSOC 2007: 169-180. DOI: 10.1007/978-3-540-74974-5_-14
- Guido Governatori and Zoran Milosevic and Shazia W. Sadiq. (2006). Compliance checking between business processes and business contracts. EDOC 2006: 221-232. DOI: 10.1109/EDOC.2006.22
- Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, Zohar Manna. (2005). LOLA: Runtime Monitoring of Synchronous Systems. TIME 2005: 166-174. DOI: 10.1109/TIME.2005.26
- Hubert Garavel and Radu Mateescu. (2004). SEQ.OPEN: A Tool for Efficient Trace-Based Verification. SPIN 2004: 151-157. DOI: 10.1007/978-3-540-24732-6_11
- Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, Oleg Sokolsky. (2004). Java-MaC: A Run-Time Assurance Approach for Java Programs. FMSD 2004: 129-155. DOI: 10.1023/B:FORM.0000017719.43755.7c

Automata, Logic, Formal Methods

- Michael Huth, Mark Ryan. (2004). Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press. ISBN: 978-0521543101
- Amir Pnueli. (1977). The Temporal logic of programs. FOCS18: 46-57. DOI: 10.1109/SFCS.1977.32

Index

ActionEvent, 153
Adder, 29
aggregation function, 57
ambient intelligence, 206
annotation, 253
Ant, 12
ApplyFunction, 37, *280*
ApplyFunctionPartial, 109, 151, *280*
arity, *280*
Azrael (library), 171

Backus-Naur Form (BNF), 247
Bags, 79, *280*
    AnyElement, 183, *279*
    ApplyToAll, 81, 182, *280*
    Contains, 79, *282*
    Explode, *284*
    FilterElements, 81, *284*
    GetSize, 79, *286*
    Product, *293*
    RunOn, 83, 190, *296*
    ToArray, 82, *299*
    ToList, 82, *299*
    ToSet, 82, *299*
Berners-Lee, Tim, xii
BigInteger, 174
BinaryFunction, 225, *281*
Bison (parser), 247
BitmapJFrame, 159
BlackHole, 32, *281*
BlowTuples, 128
Booleans, 35, *281*
    And, 50, *279*
    Implies, *287*
    Not, 36, *290*
    Or, *291*
Borealis, 3
Builds (annotation), 253
Bullwinkle (parser), 247

calculator, 255
Call, *281*
CallbackSink, *281*

Cayuga, 3
`ChangeEvent`, 155
`ChangeListener`, 154
`clean` (annotation), 259
closed (chain), *281*
`Connector`, 19, 223, *281*
`ConnectorException`, 32
`Constant`, 266, *281*
`ContextAssignment`, 136, *282*
`ContextVariable`, 136, *282*
`CountDecimate`, 62, *282*
CSV (file format), 123
`Cumulate`, 47, *283*
`CumulativeFunction`, *283*

decimation, 62
`Deserialize`, *283*
distributed computing, 168
domain-specific language, 245
`Doubler`, 19
`DrawPlot`, 158

Eclipse (IDE), 7
`Equals`, *283*
Esper, 3, 245
event, 1
`Eventually`, *283*
`Exists`, 140, *283*
`ExpandAsColumns`, 128, *284*

`FetchAttribute`, 126
file
    reading from, 94
`Filter`, 64, 152, *284*
`FindPattern`, 100, *284*
finite-state machine, 130
first-order logic, 140
`ForAll`, 140, 183, *285*
`Fork`, 43, *285*
`Freeze`, *285*
front, 23, *285*
`Function`, 35, *285*
    `duplicate`, 221
    `evaluate`, 222
    `evaluatePartial`, 226
    `getInputArity`, 221
    `getInputTypesFor`, 222
    `getOutputArity`, 221
    `getOutputTypeFor`, 223
`FunctionTransition`, 132, *286*
`FunctionTree`, 41, *286*

`GetWidgetValue`, *286*
GitBook, xi
GitHub, 10, 12
`Globally`, 144, *286*
grammar, 247
`GrammarObjectBuilder`, 251
`GroupProcessor`, 59, 162, *287*
GUI, 153

Heron's formula, 226
HL7, 194
HTTP, 101, 168
`HttpDownstreamGateway`, 168
`HttpGet`, 101, *287*
`HttpUpstreamGateway`, 168

`IdentityFunction`, *287*
`IfThenElse`, 77, *287*
`InputStream`, 94
`Insert`, *287*
`IntegerDivision`, 37, 61
`IsEven`, 58, 72
`Iterator`, 95

J-Lo, 3
JavaMOP, 3
JDBC, 128
`JdbcSource`, 128, *288*

JPathFunction, 178, *288*
JSON, 171, 176
JsonDeserializeString, 171
JsonElement, 177, *288*
JsonSerializeString, 171

KeepLast, 73, *289*

landmark query, 189
LARVA, 3
Leibniz formula, 77
Limit, 167, 208, *289*
Linear Temporal Logic (LTL), 5, 143
LINQ, 3
ListenerSource, 153, *289*
Lists
  Pack, 86, *291*
  TimePack, *299*
  Unpack, 87, *301*
LogFire, 3

Map (interface), 89, 125
Maps, 89
  ArrayPutInto, 90, *280*
  Get, 89, *286*
  PutInto, 89, *294*
  Values, 90, *301*
MarQ, 3
MergeTuples, 127
MFOTL, 246
monitoring
  enforcement, 151
MonPoly, 3, 246
MooreMachine, 131, 199, 209, *289*
MTNP, 156
Multiplex, *290*
Multiset, 217, *290*
mutable object, 45
mutator processor, 85

named pipe, 98
Next, 147, *290*
NIALM, 207
NthElement, 85
Numbers, 35, *290*
  AbsoluteValue, *279*
  Addition, *279*
  Division, *283*
  IsEven, *288*
  IsGreaterThan, *288*
  Maximum, *289*
  Minimum, *289*
  Multiplication, *290*
  NumberCast, 95, 183, *290*
  Power, *292*
  Signum, *296*
  SquareRoot, *298*
  Subtraction, *298*

offline processing, 2
online processing, 2

palettes, 8
Pandoc, xi
ParseJson, 177, *291*
ParseXml, 180, *291*
parsing tree, 250
Passthrough, 31, 264
peak (signal), 163
PeakFinder, *291*
PeakFinderLocalMaximum, 163, 208, 292
PeakFinderTravelRise, 164, 292
Perl, 3
Persist, 167, *292*
PHP, 3
Pingus (video game), 212
pipe
  colour coding, 274

planetary encounter, 201
PlateauFinder, 165, 209, *292*
plots, 156
  scatterplot, 159
PoET, 3
Polish notation, 251
pop (annotation), 257
Prefix, 188, *292*
Print, 29, 98, *292*
Processor, *292*
  duplicate, 116, 230
  duplicateInto, 241
  getContext, 230
  getId, 112, 230
  getInputArity, 229
  getInputTypesFor, 231
  getOutputArity, 229
  getOutputTypeFor, 231
  getPullableOutput, 17, 230
  getPushableInput, 27, 230
  reset, 56, 111
  setContext, 230
  setPullableInput, 230
  setPushableOutput, 230
processor, 15
  arity, 16, 59
  context, 114, 134, 182, *282*
  duplication, 115, 234
  ID, 112
  uniform, 89
pull mode, 16, 91, *293*
  hard pulling, 103
  soft pulling, 105
Pullable, 17, *293*
  hasNext, 95
  next, 95
  NextStatus, 106
  pull, 18
PullableException, 21, 41
Pump, 92, *293*
push mode, 27, 88, 91, *294*
Pushable, 27, *294*
  notifyEndOfTrace, 75
  push, 27
Python, 3

quantifier, 140
QueueSink, 27, *295*
QueueSource, 8, 17, *295*

RaiseArity, *295*
Randomize, 165, *295*
ReadInputStream, *295*
ReadLines, 94, 201, *296*
ReadStreamString, 97
ReadStringStream, *296*
regular expression, 100, 248
Runnable (interface), 92
running average, 49, 58, 269
runtime verification, 1, 212

ScalarIntoTuple, 127
serialization, 171
Serialize, *296*
Sets, 35, *296*
  IsSubsetOrEqual, 84, *288*
  PutInto, 85, *294*
  PutIntoNew, 85, *295*
Siddhi, 3
singleton, 36, 235
Sink, *297*
SinkLast, *297*
Slice, 69, *297*
Smooth, 203, *297*
snapshot query, 188
Software Heritage (platform), xii
Source, *297*
Splice, 201, *297*
SpoX, 3

SQL, 128
standard input, 97
stateful processor, 47
stream, 1
StreamBase SQL, 3
`StreamReader`, 98
`StreamVariable`, 266, *298*
`Strings`, *298*
  `Concat`, *281*
  `Contains`, *282*
  `EndsWith`, *283*
  `FindRegex`, *285*
  `Matches`, *289*
  `SplitString`, 202, *298*
  `StartsWith`, *298*
  `ToString`, *300*
`Stutter`, *298*
Swing (library), 153
`SynchronousProcessor`, 231, *296*
  `compute`, 231

`Table`, 156
`Tank`, 93, *298*
`TankLast`, *299*
TelegraphCQ, 3
TeSSLA, 3
`Thread`, 92
`Threshold`, 166, 208, *299*
`TimeDecimate`, *299*
`ToImageIcon`, 156, *300*
tokenization, 99
Tracematches, 3
`Transition`, 132, *300*
`TransitionOtherwise`, 138, *300*
`Trim`, 11, 51, *300*
`Tuple`, *300*
tuple, 124
`TupleFeeder`, 124, *300*
`TurnInto`, 50, *300*
tutor, 11
twin primes, 173

`UnaryFunction`, 181, 224, *300*
  `getValue`, 181, 225
uniform processor, 51
`Until`, 148, *301*
`UpdateTable`, 156
`UpdateTableArray`, 156
`UpdateTableStream`, 156, 161, 203

`VariableStutter`, 161, *301*
`Variant`, 234, *301*
VoltDB, 3

`WidgetSink`, 155
`Window`, 54, *301*
window
  hopping, 64
  sliding, 54
`WindowFunction`, *302*
`WriteOutputStream`, *302*

XML, 179
`XmlElement`, 180, *302*
`XPathFunction`, *302*

Yacc, 247

IN THE SAME COLLECTION

LA MODÉLISATION PAR ÉQUATIONS STRUCTURELLES AVEC MPLUS
Pier-Olivier Caron
ISBN 978-2-7605-4972-2, 280 pages

CONCEPTS ET OUTILS DES SONDAGES WEB
Introduction à LimeSurvey et SurveyMonkey
Michel Plaisent, Lili Zheng, Mariem Khadhraoui et Prosper Bernard
ISBN-978-2-7605-5002-5, 164 pages

L'ANALYSE DES DONNÉES DE SONDAGE AVEC SPSS
Un guide d'introduction
Lili Zheng, Michel Plaisent, Cataldo Zuccaro, Prosper Bernard, Naoufel Daghfous et Sylvain Favreau
ISBN-978-2-7605-4914-2, 144 pages

MESURE ET ÉVALUATION DES COMPÉTENCES EN ÉDUCATION MÉDICALE
Regards actuels et prospectifs
Sous la direction de Eric Dionne et Isabelle Raîche
ISBN-978-2-7605-4796-4, 268 pages

INTRODUCTION À LA MODÉLISATION D'ÉQUATIONS STRUCTURELLES
AMOS dans la recherche en gestion
Lili Zheng, Michel Plaisent, Cataldo Zuccaro et Prosper Bernard
ISBN-978-2-7605-4738-4, 118 pages

CONSTRUIRE DES GRILLES D'ÉVALUATION DESCRIPTIVES AU COLLÉGIAL
Guide d'élaboration et exemples de grille
France Côté
ISBN-978-2-7605-4101-6, 192 pages

DES MÉCANISMES POUR ASSURER LA VALIDITÉ DE L'INTERPRÉTATION DE LA MESURE EN ÉDUCATION, Volume 1 – LA MESURE
Sous la direction de Gilles Raîche, Karine Paquette-Côté et David Magis
Avec la collaboration de Diane Leduc et d'Hélène Meunier
ISBN-978-2-7605-2685-3, 148 pages

DES MÉCANISMES POUR ASSURER LA VALIDITÉ DE L'INTERPRÉTATION DE LA MESURE EN ÉDUCATION, Volume 2 – L'ÉVALUATION
Sous la direction de Gilles Raîche, Karine Paquette-Côté et David Magis
Avec la collaboration de Diane Leduc et d'Hélène Meunier
ISBN-978-2-7605-2687-7, 178 pages

DES MÉCANISMES POUR ASSURER LA VALIDITÉ DE L'INTERPRÉTATION DE LA MESURE EN ÉDUCATION, Volume 3 – ASPECTS PRATIQUES
Sous la direction de Gilles Raîche, Pascal Ndinga et Hélène Meunier
ISBN-978-2-7605-3593-0, 172 pages

Event logs and event streams can be found in software systems of very diverse kinds. For instance, workflow management systems and ERP platforms produce event logs in some common format based on XML.

Financial transaction systems also keep a log of their operations in some standardized and documented format, as is the case for web servers such as Apache and Microsoft IIS. Network monitors also receive streams of packets whose various headers and fields can be analyzed. Recently, even the world of video games has seen an increasing trend towards the logging of players' realtime activities.

Analyzing the wealth of information contained in these logs can serve multiple purposes. Business process logs can be used to reconstruct a workflow based on a sample of its possible executions; financial database logs can be audited for compliance to regulations; suspicious or malicious activity can be detected by studying patterns in network or server logs. However, the available tools to process logs or streams of events are often large systems that are hard to setup, and even simple examples seem needlessly complicated.

In this book, you will learn about BeepBeep, a versatile Java library intended to make the processing of event streams both fun and simple.

Through more than a hundred simple, illustrated code examples, you will see how running event processing tasks can be done in just a few lines of code—and what is more, code that you actually understand. From generating plots to computing statistics and evaluating temporal logic specifications, BeepBeep can prove a handy addition to a developer's toolbox.

SYLVAIN HALLÉ is a Full Professor in the Department of Computer Science and Mathematics at Université du Québec à Chicoutimi, Canada, and is the current holder of the Canada Research Chair on Software Specification, Testing and Verification. A free software enthusiast, he has been the recipient of multiple awards for his research on software testing, runtime monitoring and event log analysis.