FITSL: a Language for Directed Exploration and Analysis of Sequence Data

by

Eric J.P. Mumpower

Bachelor of Science, Massachusetts Institute of Technology (2003)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

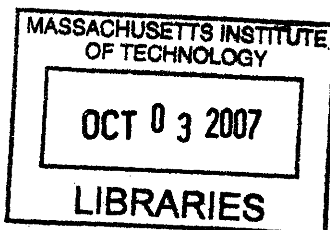
February 2007

© Massachusetts Institute of Technology 2007. All rights reserved, except that This work is licensed under the Creative Commons Attribution 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Author ....
Department of Electrical Engineering and Computer Science
November 8, 2006

Certified by ....
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 0 3 2007
LIBRARIES

ARCHIVES

FITSL: a Language for Directed Exploration and Analysis of Sequence Data

by

Eric J.P. Mumpower

Submitted to the Department of Electrical Engineering and Computer Science
on November 8, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a sequence-data processing toolkit for analysis of Intelligent Tutoring System (ITS) log data, that unlike other tools allows directed exploration of sequence patterns. This system provides a powerful yet straightforward abstraction for sequence-data processing, and a set of high-level manipulation primitives which allow arbitrarily complex transformations of such data. Using this language, very sophisticated queries can be performed using only a few lines of code. Furthermore, queries can be constructed interactively, allowing for rapid development, refinement, and comparison of hypotheses. Importantly, this system is not limited to ITS logs, but is equally applicable to the manipulation of any form of (potentially multidimensional) sequence data.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

I would like to express my deepest appreciation to the following:

Harold Abelson, Class of 1922 Professor of Computer Science and Engineering, for his counsel, support, patience, and encouragement throughout this endeavor

Anne Hunter, MIT EECS, *condicio sine qua non*

The many people who contributed to the Cambridge-MIT Institute's Intelligent Book project, without whom this thesis would never have happened

Gerald Sussman, Chris Hanson, and Selene Victor, for their wisdom and unflagging assistance in their respective domains

Jacob Beal, for his seasoned perspective and advice

Edi Weitz, for his excellent Common Lisp regular-expression library

My loved ones and my friends for their encouragement and support

And, most particularly, my wife, Johanna Bobrow, for her inspiration and love, her unflagging faith, and her tireless support of my studies.

Contents

1 Scenario **11**

1.1 Introduction 11

1.2 Motivation 11

1.3 Scenario: Basic Usage 13

1.3.1 Experimental Data 13

1.3.2 FITSL Data-Processing Paradigm 14

1.3.3 A FITSL Experiment 15

1.3.4 FITSL Review 22

2 Related Work **25**

2.1 Data Mining and Sequence Mining 25

2.2 Programmable Database Trigger Rules 26

2.3 Agrawal's *Shape Description Language* 27

3 Language **29**

3.1 Processing Paradigm 29

3.2 Language Overview 30

3.3 Core Operators 31

3.3.1 Splitting Modifiers 32

3.3.2 Basic Filters 35

3.3.3 Extract 37

3.3.4 Flatten 38

3.3.5 Join 40

3.4 Advanced Filters 43
3.4.1 Difference Filters 43
3.4.2 Higher-order Filters 44
3.4.3 Regexp Filters 45
3.4.4 Whole-stream Filters 49
3.4.5 Composite Filters 50

4 Other Examples **53**
4.1 Extensibility and Interoperation 53
4.1.1 Language Extensibility: User-Provided Functions 54
4.1.2 Translating Data into Standard Lisp Structures 55
4.2 Demonstration of Language Features 56
4.2.1 The **`extract`** Primitive 56
4.2.2 Basic Difference Filters 57
4.2.3 The **`flatten`** Primitive 58
4.2.4 Data Exploration with FITSL 59
4.3 Higher-Order Filters 62
4.3.1 Recognizing Binary Search 63
4.3.2 Data Analysis with FITSL 64

5 Implementation **67**
5.1 Design Criteria 67
5.1.1 Overall Language Considerations 67
5.1.2 Modes of Operation 68
5.2 Choice of Implementation Language 68
5.3 Various Implementation Decisions 69
5.3.1 Data Manipulation Paradigm 69
5.3.2 Regular Expression Filters 70
5.3.3 Data Representation 71
5.3.4 Modular Data Import/Export Functionality 73
5.4 Summary 74

6 Future Work **75**

6.1 Performance Improvements 75

6.1.1 Regular Expression Performance 75

6.1.2 Enhancements to **unsplit** 76

6.1.3 Performance of **extract** 77

6.2 Minor Feature Enhancements 77

6.3 Major Functional Additions 78

Chapter 1

Scenario

1.1 Introduction

This thesis describes the Filtering Interactive Time-Sequence Language (FITSL), a sequence-data processing toolkit for analysis of Intelligent Tutoring System (ITS) log data, that unlike other tools allows directed exploration of sequence patterns. FITSL provides a powerful yet straightforward abstraction for sequence-data processing, and a set of high-level manipulation primitives which allow arbitrarily complex transformations of such data. Using this language, very sophisticated queries can be performed using only a few lines of code. Furthermore, FITSL queries can be constructed interactively from a Lisp interpreter, allowing for rapid development, refinement, and comparison of hypotheses. FITSL was developed in response to the apparent lack of similar tools; as is explained in the next section, this lack was encountered in the course of analyzing patterns in ITS log data.

1.2 Motivation

Decades of educational computing research have resulted in the development of many Intelligent Tutoring Systems (ITSes). Frequently, such work aims to emulate a human tutor, in an attempt to reap some of the two-sigma gain in learning [7] which results when students work one-on-one with a teacher.

In part, deploying such a system is attractive because an ITS allows an educator to greatly amplify their ability to reach students. A person can only provide one-on-one tutelage to a handful of students in the course of an hour. In contrast, an ITS (running on one inexpensive server) can manage simultaneous sessions with hundreds or even thousands of students.

This increased ability of educators to convey information to students, however, has not been accompanied by a equal increase in educators' ability to analyze their students' experiences. Just as a human tutor receives direct and indirect feedback regarding their effectiveness and the ways that the students choose to engage with their teaching, an ITS can record a richly annotated transcript of a student's interaction with the system. Manual analysis of such logs can be quite revealing, both with regards to the surface-level quantitative statistics, and in terms of subtler behavior patterns, such as simply ignoring advice given by the system [29] or "gaming the system" [5, 24]. Obviously, the potential number of ITS students prohibits manual analysis of every single ITS transcript. Technology has given us this embarrassment of riches by allowing massively parallel use of ITSes; it is natural to turn once again to computers for the solution to this problem.

Unfortunately, at the outset of this project, it was discovered that no existing tools are well-suited to a directed analysis of annotated time-sequence data (as will be discussed in Chapter 2). A tailor-made data-analysis tool could be written to test any given hypothesis, but such a bespoke approach can be especially ill-suited to developing/refining a query. Furthermore, writing one program per hypothesis grows quickly untenable. Thus, a system was designed and implemented to meet this need: the Filtering Interactive Time-Sequence Language (FITSL).

It will be shown that FITSL allows the concise expression of arbitrary sequence-data manipulations, and that the language operators can be combined to provide any given mutation of sequence data. Furthermore, it will be demonstrated that the data-manipulation engine provided by FITSL is reasonably efficient, and that basic data-manipulation operations can be composed and tested by the user with ease.

The next section presents a demonstration of the power and convenience of FITSL

in a typical usage scenario.

1.3 Scenario: Basic Usage

Alice is an educational researcher with log data from several classes using an ITS. She is exploring the hypothesis that students who review material more frequently also take a larger number of trials before getting correct answers. Presumably, both behaviors reflect a lower mastery of the material. To test this hypothesis, Alice first filters the data to extract and measure lesson-reviewing behavior, and uses a series of filters to measure the number of submissions before a correct answer. This second step requires several sub-steps: removing dummy records and problems that never got a correct submission, removing students who presumably dropped the class, and finally measuring the number of submissions before getting a correct answer. Finally, Alice joins the two sets of data points produced by the filters and measures both the overall correlation and the per-class correlations between lesson-reviewing behavior and submissions before a correct answer. While she discovers a moderate correlation for all the students taken together, she also finds that the per-class correlation for the different classes varies enormously.

The ensuing sections present the above experiment in greater detail. Section 1.3.1 describes the real-world dataset used in the experiment, and Section 1.3.2 introduces the data-processing paradigm used by FITSL. The various stages of experiment itself are described in Section 1.3.3. Finally, Section 1.3.4 gives a brief summary of FITSL's features and strengths.

1.3.1 Experimental Data

MIT's main introductory course in Computer Science is *Structure and Interpretation of Computer Programs* (also known as 6.001 [1]). An ITS presenting a full online version of this course has been authored, including texts, lectures, and interactive homework assignments. This tutor was used by MIT students enrolled in *Introduction to Scheme Programming* (6.188); students at the University of Queensland, Australia;

Class	Students in class	Designation
University of Queensland (2005)	320	UQ
CETI (2005)	43	CETI
CETI (2006)	37	CETI2
MIT 6.188 (2006)	24	6.188

Table 1.1: 6.001 XTutor Class Information

and students participating in the China Educational Technology Initiative (CETI)[1]. Data was collected from 424 students in four different classes, divided as can be seen in Table 1.1. Transcripts of all student interactions with the 6.001 tutor were collected, in the form of time sequences containing, in total, over 360,000 distinct annotated events. While FITSL can import data from CSV, ARFF (WEKA [28]), and Matlab [17] file formats, the 6.001 XTutor [19] logs each event datum as a textual representation of a Lisp data structure; hence, a special-purpose XTutor-log module is used to import the data into FITSL.

1.3.2 FITSL Data-Processing Paradigm

FITSL is intended to deal with the processing of sequential data streams. As such, while its fundamental unit of data processing is an annotated event object (akin to a multi-field database record), all FITSL primitives operate on a "streamset" – a set of event streams. In this paradigm, each event stream is an ordered sequence of events, and a streamset consists of an unordered set of these event streams. Generally, streamsets are partitioned into their constituent streams on the basis of the values of one or more attribute. For example, if one had a streamset split by student ID, that streamset would contain one stream per student, where each stream contained the time-ordered list of that student's event objects.

[1]The CETI program [8] introduces Chinese schools to various educational technologies which have been created at MIT, to "promote cultural exchange between Chinese and American students." Students in Australia and China used the 6.001 XTutor server as part of regular computer science courses offered by CETI and the University of Queensland.

1.3.3 A FITSL Experiment

Here is presented a data-exploration experiment in which Alice investigates student use of the 6.001 tutor. In Section 1.3, this experiment was outlined briefly; the following sub-sections explain each of the logical steps of the experiment in detail.

Experiment Background

Students using the 6.001 ITS can review lesson material while working on homework assignment problems. When finished with a problem, they electronically "hand them in" for grading. In most cases prior to the final hand-in, however, students can ask the ITS to check their answers, indicating only whether they are right or wrong. Students may then revise their answers and re-submit them as many times as they like prior to final hand-in.

Statement of Hypothesis

Alice believes that when a student is having more difficulty than usual with the course material, they will submit a larger number of wrong answers for checking, prior to finding a correct answer. She knows that, when having trouble with a homework assignment, some students will review lesson materials in an effort to improve their understanding of the topic. Furthermore, the relative incidence of such behavior will vary from student to student. Alice decides to explore the hypothesis that, among students, there is a positive correlation between the incidence of such lesson-reviewing behavior and the number of wrong answers a student submits prior to getting a correct answer on a problem. In short, she believes both behaviors occur in proportion to the difficulty a student has with the course-work.

Loading Data

Alice begins her work by invoking her Common Lisp [12] interpreter and loading the FITSL packages. As shown in Figure 1-1, she then loads the 6.001 dataset[2] and

[2] A log-parsing module (customized for the XTutor log format) was previously used to load all 6.001 student data into FITSL; the resulting dataset was then dumped to disk in the special `.fd`

applies a standard filter to the dataset (which weeds out extraneous log entries and performs some standard annotations).

```
(progn
  (setf everyone (load-data 'fd "everyone.fd"))

  (setf everyone-data
    (xtutor-apply-standard-filters everyone)))

;; operation runtime was 1:22 (mm:ss)
;; operation runtime was 2 seconds
;; operation runtime was 1 second
;; operation runtime was 14 seconds
;; operation runtime was 15 seconds

;; TOTAL runtime was 1:54 (mm:ss)

==> #<ESTREAM-SET @ #x7808eba2>
 ... Estream-set containing 334058 events of type EVENT-XTUTOR
 ... with native attribs:    (CLASS TIME USERID ENTRYTYPE PROBLEMNAME SUBMITTED
                             HINTS CODING-HINTS ANSWERS)
 ... and annotated attribs:  (FRACRIGHT PROBLEM-SUBMISSION)
 ...  and split by: (USERID CLASS)
 ...  into 424 discrete streams
```

Figure 1-1: Initial Loading and Annotation of Data

Lesson-Reviewing Behavior

She begins by annotating each student's data to indicate the times when that student is demonstrating lesson-reviewing behavior. She decides to define such behavior as times when a student submits a wrong answer to some homework problem (`gotwrong`), then accesses the lesson material, and finally returns to that *same* homework problem (`returnedto`). She uses the annotation `reviewing-lesson` to label such transcript episodes. We can see her encoding of this pattern in Figure 1-2; FITSL uses an S-expression syntax to encode regular-expression patterns of events[3].

(FITSL Data) format. On the system used for these experiments, parsing this dataset from the logs takes about five times as long as loading from the `.fd` file.

[3]In particular, it uses a variant of the sexp-regexp syntax used by the CL-PPCRE package [26], which has been enhanced to allow the use of FITSL event-datum objects as regexp atoms. Event regexp-atoms may be specified so they only match event objects satisfying given criteria. Also, as described in this example, annotation of regular-expression portions may be specified in the sexp.

```
(setf students-reviewing-lesson
 (filter everyone-data
  'regexp
  '(:ANNOTATE reviewing-lesson
     (:SEQUENCE
       (:ANNOTATE gotwrong (:EVENT (problem-submission)
                                   (fracright /=     nil 1.0)))
       (:ATLEAST 1 (:EVENT entrytype eq     nil :LESSON-PART))
       (:ANNOTATE returnedto (:EVENT (problem-submission)
                                     (problemname string= gotwrong)))))
  :keepnulls t))

;; operation runtime was 36 seconds

;; TOTAL runtime was 36 seconds

==> #<ESTREAM-SET @ #x744b9cfa>
 ... Estream-set containing 334058 events of type EVENT-XTUTOR
 ... with native attribs:    (CLASS TIME USERID ENTRYTYPE PROBLEMNAME SUBMITTED
                              HINTS CODING-HINTS ANSWERS)
 ... and annotated attribs:  (FRACRIGHT PROBLEM-SUBMISSION GOTWRONG
                              REVIEWING-LESSON RETURNEDTO)
 ...  and split by: (USERID CLASS)
 ...  into 424 discrete streams
```

Figure 1-2: Regular Expression Filter

FITSL provides a powerful mechanism for annotating the structure of sequences a matching regular expression (regexp): the sequences matching each `:ANNOTATE` form will be annotated with the specified symbol. Thus, in Figure 1-2, there are clauses for performing annotations named `reviewing-lesson`, `gotwrong`, and `returnedto`. Furthermore, when scanning for pattern matches within a sequence, the sequence matching each `:ANNOTATE` form (and each `:REGISTER` form) is captured in a register[4]. Thereafter, any `:EVENT` expression can make a back-reference to the attributes of the first event stored in that register. For example, in Figure 1-2 we can see that the event being annotated as `returnedto` must be a problem-submission sharing the same `problemname` as the initial (`gotwrong`) event[5]. (For more explanation of FITSL's regexp filter, see Section 3.4.3.)

[4]This is in direct parallel to the numbered regexp capture-groups and back-reference mechanism provided by the Unix **sed** tool [23] and Perl [25].

[5]I.e. XTutor problem names are unique strings, and it must have the same problem name as tested using the Lisp string-equality primitive **string=**.

For each unique student, Alice then calculates the percentage of that student's log entries which were annotated as `reviewing-lesson`. As shown in Figure 1-3, this is accomplished using the **`flatten`** primitive, which takes each stream in a streamset and condenses it into a single event of a new type. This condensation, or *flattening*, occurs by extracting (in order, from the stream) the values of one (or more) attribute(s) into one (or more) vector(s), and using a flattening procedure[6] to turn those vectors into a single value.

The operation of **`flatten`** might be best understood by way of example. In Figure 1-3, the input streamset `students-reviewing-lesson` contains one stream per unique student, and the **`flatten`** expression calculates the percentage of the events (**`percentage-true`**) in each student's stream which have been annotated as `reviewing-lesson`. Thus, a new streamset (`percentage-reviewing`) is produced, containing one event datum per student, where each event object is given appropriate values for the attributes `class`, `userid`, and `percentage-true--reviewing-lesson`.

```
(setf percentage-reviewing
  (flatten students-reviewing-lesson
           :by '(class userid)
           :fn #'percentage-true
           :attribs 'reviewing-lesson))

;; operation runtime was <1 second
;; operation runtime was 45 seconds

;; TOTAL runtime was 45 seconds

==> #<ESTREAM-SET @ #x7a884c32>
 ... Estream-set containing 424 events of type FLATTEN-TYPE-6
 ... with native attribs:    (CLASS USERID)
 ... and annotated attribs:  (PERCENTAGE-TRUE--REVIEWING-LESSON)
 ...  and split by: (CLASS USERID)
 ...  into 424 discrete streams
```

Figure 1-3: Applying the Flatten Filter

At this point, Alice has synthesized a new streamset (`percentage-reviewing`) indicating what proportion of their ITS interactions each student spends in behav-

[6]For example, the flattening procedures **`tally`**, **`mean`**, **`percentage-true`**, **`count-true`**, **`count-until-predicate`** are among those provided by FITSL, but users may write other flatteners in Lisp as desired.

ior matching her `reviewing-lesson` criteria. Next, she must extract information regarding each student's average number of wrong-answer submissions prior to a correct answer.

Number of Submissions

Each of FITSL's elementary data-manipulation operations may be regarded as a filter applied to an existing streamset. In the next step (Figure 1-4), Alice cascades a sequence of FITSL filters to achieve the desired result.

```
(setf active-students
  (let ((just-problems (filter-if-not everyone-data 'problem-submission)))
    (filter-chain just-problems
      (f 'flatten
          :by '(userid class problemname)
          :fn (make-thunk-1 #'count-until-predicate
                            :predicate (make-thunk-1 #'equal 1.0))
          :attribs   'fracright
          :flatten-to 'submissions-until-right)

      (f 'test #'not-null :attrib 'submissions-until-right)

      (f 'split-only 'userid)

      (f 'wholestream
         (comp '(lambda (stream) (> (length stream) 5)))))))

;; operation runtime was <1 second
;; operation runtime was 2:11 (mm:ss)
;; operation runtime was <1 second
;; operation runtime was 1:27 (mm:ss)
;; operation runtime was <1 second
;; TOTAL runtime was 3:38 (mm:ss)
==> #<ESTREAM-SET @ #x7cedfc7a>
 ... Estream-set containing 25094 events of type FLATTEN-TYPE-10
 ... with native attribs:    (USERID CLASS PROBLEMNAME)
 ... and annotated attribs:  (SUBMISSIONS-UNTIL-RIGHT)
 ...  and split by: (USERID)
 ...  into 373 discrete streams
```

Figure 1-4: Counting Submissions-Until-Right

To begin, she uses the **filter-if-not** tool[7] to remove all non-problem-submission events from her original dataset. She then flattens her dataset by student and problem, calculating the number of submissions each student makes on each problem before they get a correct answer[8]; this flattening produces one datum per student-problem pair. The next stage in the filter-chain discards the data corresponding to problem-work where the student never submitted a correct answer at all. The penultimate stage of the filter-chain re-splits the data by user, and the final stage elides any students who submitted correct answers for fewer than five problems[9].

```
(setf avg-student-perproblem-subs
  (flatten active-students
           :by '(class userid)
           :fn #'statistics:mean
           :attribs 'submissions-until-right
           :unsplit t))

;; operation runtime was <1 second
;; operation runtime was 1 second

;; TOTAL runtime was 1 second

==> #<ESTREAM-SET @ #x7c3866fa>
 ... Estream-set containing 373 events of type FLATTEN-TYPE-15
 ... with native attribs:    (CLASS USERID)
 ... and annotated attribs:  (MEAN--SUBMISSIONS-UNTIL-RIGHT)
 ...  and split by: NIL
 ...  into 1 discrete stream (of length 373)
```

Figure 1-5: Extracting Students' Average Submissions-Until-Right

Thus, after Figure 1-4 has been evaluated, `active-students` contains one stream per student, where each stream contains one datum for each of the problems that student completed, annotated with `submissions-until-right`, the number of tries it took that student until they submitted a correct answer for that problem. For each

[7]FITSL includes some higher-level composite/shorthand tools, such as **filter-if-not** and **split-users-and-tag-event-durations**; users may create more such tools as desired.

[8]The **make-thunk-1** procedure returns a one-argument wrapper function for any given function, passing along any specified extra arguments. Thus, if one takes the function returned by `(make-thunk-1 #'eq 'an-arg)` and applies it to the value 2.5, the value of the expression `(eq 2.5 'an-arg)` will be evaluated and returned.

[9]There are over 110 problems; on average, students submitted correct answers for 63 problems. Students who submitted fewer than five correct answers are presumed to have dropped the class.

student in `active-students`, among the problems that student completed, Alice now (in Figure 1-5) computes the mean number of `submissions-until-right`. In other words, Alice computes each student's average number of submissions prior to getting a correct answer.

As this is the second statistic Alice sought, she can now proceed to assessing the degree of correlation between these statistics.

Correlating Results

So far, Alice has extracted the desired statistics from her dataset, but at the moment these statistics are stored in two separate streamsets. As can be seen in Figure 1-6, Alice now cross-references these two streamsets using the FITSL **`join`** primitive (similar in function to the SQL [10] join command).

```
(setf xref-results
  (filter-chain avg-student-perproblem-subs
       (f 'join
           :vs percentage-reviewing
           :by   '(class userid)
           :with '(percentage-true--reviewing-lesson))))

;; operation runtime was <1 second
;; TOTAL runtime was <1 second
==> #<ESTREAM-SET @ #x7c3ce912>
 ... Estream-set containing 373 events of type FLATTEN-TYPE-15
 ... with native attribs:    (CLASS USERID)
 ... and annotated attribs:  (MEAN--SUBMISSIONS-UNTIL-RIGHT
                              PERCENTAGE-TRUE--REVIEWING-LESSON)
 ...  and split by: NIL
 ...  into 1 discrete stream (of length 373)
```

Figure 1-6: Cross-Referencing Lesson-Reviewing and Number of Submissions

The filter-chain in Figure 1-6 results in a streamset containing one stream, which contains one element per student. Each student's element contains their values for both the `percentage-true--reviewing-lesson` and the `submissions-until-right` statistics we extracted in the previous two sub-sections. Alice is finally in a position to evaluate the overall correlation of these statistics, and does so in Figure 1-7.

```
(show-correlation
 (->xy xref-results :attribs '(percentage-true--reviewing-lesson
                               mean--submissions-until-right)))

** This is a statistically significant correlation!
;; TOTAL runtime was <1 second
==> ((QUERY::R . 0.14395142) (QUERY::P . 0.0052480423))
```

Figure 1-7: Measuring Overall Correlation

From this, she sees that there is, indeed, a moderate correlation between lesson-reviewing behavior and the number of submissions before a correct answer (using Pearson's correlation, $r = 0.144$), and that she may have confidence in this correlation as her p-value[10] (0.00525) is statistically significant.

Alice has just calculated the overall correlation of two attributes among the events in this streamset. In Figure 1-8, the great expressive power of FITSL is demonstrated by the ease with which one can independently calculate this correlation for *every* stream in this streamset. Here, Alice sees that, within classes, the degree of correlation varies wildly; in particular, while there is a strong and significant correlation amongst the CETI students, these statistics are *almost completely independent* amongst the 320 University of Queensland students.

1.3.4 FITSL Review

This is just one example of the applications to which FITSL is suited. This experiment alone could have been taken in many different directions, and there are an vast range of experiments which one might perform on a dataset; additional examples are given in Chapter 4. As it's embedded in Lisp, FITSL provides a framework into which any user-written filter can be integrated. Also, FITSL can manipulate *any* set of sequential data points, such as weather and stock-market data, or server logs. What's more, FITSL is not merely an tool for interactive use; it's also well-suited for the

[10]The p-value is the probability of one's data under the null hypothesis. In this case, the p-value is the probability of having obtained this particular data set assuming that there was no correlation.

```
(progn
  (setf checking-correlation-class-dependence
    (flatten (split xref-results 'class)
             :by 'class
             :attribs '(percentage-true--reviewing-lesson
                        mean--submissions-until-right)
             :fn #'show-correlation
             :flatten-to 'conditional-correlation))
  (princ checking-correlation-class-dependence))

;; operation runtime was <1 second
** This is a statistically significant correlation!
;; operation runtime was <1 second
;; operation runtime was 1 second
;; Estream-set,
;;  containing events of type:
      <eventtype 'FLATTEN-TYPE-19>
;; with native attribs:    (CLASS)
;; and annotated attribs:  (CONDITIONAL-CORRELATION)
;;  and split by: (CLASS)
({ (CETI2) : CONDITIONAL-CORRELATION=((R . 0.14892593) (P . 0.3019788)) })
({ (UQ) : CONDITIONAL-CORRELATION=((R . 0.016508041) (P . 0.1996583)) })
({ (CETI) : CONDITIONAL-CORRELATION=((R . 0.4389962) (P . 0.009235499)) })
({ (FOR-6.188) : CONDITIONAL-CORRELATION=((R . 0.2653368) (P . 0.18317242)) })

;; TOTAL runtime was 1 second
```

Figure 1-8: Measuring Conditional Correlation, Given Class

encoding of unattended data-processing tasks.

As has been stated, FITSL differs from other existing tools in that it allows directed investigation and analysis of patterns in multidimensional time-sequence data. These patterns may be extracted using any of a number of different methods, including higher-order discrete filters[11], description via abstract regular-expression, and many forms of statistical analysis. The language also has a succinct high-level grammar which provides a powerful and intuitive paradigm for processing multiple data-streams in parallel.

FITSL minimizes the time it takes users to implement any given sequence-data analysis, allowing users to concentrate on the exploration of their ideas rather than being distracted by implementing the special-purpose query tools which might other-

[11]as will be discussed in Sections 3.4.2 and 4.3

wise be required. Furthermore, it provides abstractions and an extensible framework which greatly ease the rigors of implementing any data transformations which cannot readily be created through aggregation of existing FITSL primitives and library functions.

FITSL's interactive mode makes incremental query refinement much more convenient than is possible with any noninteractive toolkit or special-purpose applications. What's more, its data-processing engine provides high performance for most operations, making real-time interactive use practical; this enables users to incrementally develop and compare hypotheses by minimizing delay between iterations.

The following chapters give a deeper understanding of FITSL. They begin with a more detailed discussion of prior work in this area, and go on to give an in-depth treatment of the language itself. A chapter is devoted to further demonstration of the use of this toolkit, highlighting various capabilities which could not be detailed in this introduction. The closing portions of this paper detail FITSL's implementation, and outline possible directions for future work.

Chapter 2

Related Work

Investigation of computer-aided data-exploration has led to many different areas of research, including those of data mining, sequence mining, and programmable database trigger rules. While work in these areas has produced many powerful tools, they are largely unsuitable for mining preconceived sequence patterns. This chapter begins by giving an overview of each of these areas in turn, and ends with a discussion of one notable project, the Shape Description Language, which allows recognition of user-specified sequence patterns.

2.1 Data Mining and Sequence Mining

There has been a great deal of work in the area of data mining, which aims to identify correlations between attributes in a dataset (e.g. comparing students' final grades with their pretest scores). However, most of this work is inapplicable to sequence datasets, which contain time-sequence data and are likely to encode much implicit information which is only apparent when the sequence is regarded as a whole. For example, a sequential dataset might contain a timestamped list of weekly subject-mastery measurements taken from students in an ITS-based class; time-dependent information such as the overall trajectory of a student's mastery scores is invisible to traditional data-mining tools. As a result, the related field of sequence mining has been developed for analysis of such sequential data.

Table 2.1: Data Domains Explored by Existing Sequence Mining Tools

Domain	*Examples*
traversal of a finite graph	logs of user navigation through a web site
sequences of one-dimensional numeric samplings	climate data; stock market data
market-basket logs	per-user activity on an e-commerce site; credit-card or cell-phone activity logs

While sequence mining aspires to find patterns in time-sequence data, work in this area has generally focused on unsupervised pattern extraction [3, 22, 16, 20], which precludes investigation of preconceived patterns. Thus, such systems are often used for automated pattern discovery and extraction of interesting, common, or anomalous sequence features. Also, most work in sequence mining has explored data which falls within one of limited domains described in Table 2.1; ITS log transcripts lose much of their meaning when mapped into any of these domains. For all the above reasons, such sequence-mining tools are unsuitable for exploration of preconceived hypotheses regarding sequence patterns in ITS log data.

2.2 Programmable Database Trigger Rules

In the early 1990's, before sequence mining became a popular area of research, there was a fair amount of work by the DBMS research community on the topic of programmable database trigger rules [21, 18, 14, 11, 9, 27, 15]. Such rule-systems generally took the form of an enhancement to an existing database system, allowing the database to take a specified action on the arrival of a sequence of new records which satisfied a given programmable "trigger rule." These systems produced a variety of languages which allowed high-level descriptions of trigger criteria, thereby providing powerful abstractions for the encoding of match criteria for such rules.

These trigger rules recognized human-specified patterns in an arriving stream of database records. However, they were generally not used for large-scale data analysis,

but instead, they encoded programmed reactions to near-term conditions[1]. There is little indication that such languages were ever viewed as tools for the sort of larger-scale statistical exploration which would subsequently be referred to as "mining." Hence, while some of these languages possessed novel pattern-description capabilities, none of them were suited to the kind of ITS log analysis for which FITSL was eventually written.

2.3 Agrawal's *Shape Description Language*

There was, however, one sequence mining tool which is reasonably relevant to the task of ITS log analysis. SDL was a shape definition language which used a Lisp-like syntax to find patterns in a numeric single-variable (scalar) time sequence [2]. It provided a set of primitives and constructors which were equivalent in flexibility and power to a more traditional regular expression language.

Papers on SDL were first published in 1995; in 2006, this author's search for any ensuing (or related) sequence-mining work was largely fruitless. Unfortunately, with the exception of the aforementioned work on database trigger-rules, no other fields could be found with projects similar in spirit to SDL.

FITSL has borrowed one key notion from SDL: that of using a form of regular expression to describe data patterns. However, while SDL was capable of processing only solitary sequences of real numbers, FITSL extends significantly upon this. Not only does FITSL handle multidimensional data with arbitrarily-typed attribute values, but it also provides a succinct syntax for manipulating sets of multiple datastreams en masse, as will be described in Section 3.3.4.

[1]E.g. such a trigger rule could issue a stock-market "sell" order whenever a stock price first (a) went up more than 50% over its ten-week average and subsequently (b) dropped by 10% of that peak price.

Chapter 3

Language

This chapter presents a detailed description of FITSL's operators; it begins with two sections which give a broad overview of the language, discussing in turn its processing paradigm and the kinds of operations which can be performed. The last half of the chapter provides a command reference for the language, broken into a pair of sections describing the basic and advanced operators.

3.1 Processing Paradigm

As described briefly in Section 1.3.2, FITSL is a sequential-data processing language which operates on sequence sets. These sets are called *streamsets*, and each contain some number of ordered sequences of events. An event is an individual multidimensional datum, which could be used to represent, for example, an ITS log entry (bearing attributes for time, user-id, submitted-answer, and answer-score) or a weather measurement (containing values for time, station-name, temperature, and vector-wind). Sequences of events are often sorted by time, and are termed *streams.*

Streamset objects, streams, and events are themselves immutable: one cannot modify the contents of a given streamset, or the existing attributes of an event. Instead, each FITSL operator creates a transformed child of its input streamset.

Each event is a multidimensional datum, possessing various named attributes, each of which has a value. Some of these attributes were created when the event (or

its ancestor) was first imported into FITSL, and are referred to as "native attributes." The other attributes are the results of annotations which FITSL attached to some ancestral event; such annotations are referred to as "annotated attributes."[1]

Streamsets are generally split into streams according to the values of one or more attribute. For example, if a streamset had been split by student-ID and problem-number, it would contain one stream for every observed student-problem pair; each stream would contain the complete time-ordered list of a given student's events involving a given problem. It will be described in Section 3.3.4 how this splitting interacts with the language itself to allow implicit iteration over the values taken by a field; for example, to find the average time-to-completion of every student in an ITS dataset.

3.2 Language Overview

FITSL provides an all-purpose toolkit for manipulation and analysis of event sequences. The heart of the language is a set of operators which produce transformations of streamsets. One may view every operator (and every aggregation of operators) as some variety of streamset filter, each with its own effect. Many operators create annotated versions of events and/or remove certain events from the generated streamsets. Some change the ways in which events are grouped into streams, and some generate streamsets whose events differ substantially from their parents, deriving their values from selected attributes of the parent streamset. These various operators will be enumerated in the following section.

As will be demonstrated in Chapter 4, appropriate combinations of these primitives can produce arbitrary transformations of sequence datasets. Data may be partitioned, according to value, along one or more axes (e.g. by city, stock symbol, etc.), and transformations may be applied within each sub-stream resulting from this grouping. Simple queries can be expressed concisely, and primitives may be aggregated to extract much more subtle information about patterns in datasets, and to analyze this information in whatever way is desired.

[1]For more information on the annotation paradigm, see Section 5.3.3.

One should note that, with one exception, all FITSL operators take an streamset as input, and create an entirely new streamset which is a transformed copy of their input. The exception in this case is the **join** operator, which (like the other filters) creates just one output streamset, but takes two streamsets as input: a primary and an auxiliary input[2].

One should also be aware that, in general, the transformed streamset created by any filter will contain events of the same underlying eventtype as its input streamset (although the output streamset may have annotations attached to its events). The **extract** and **flatten** filters are the most important exceptions to this rule; they create streamsets containing events of an entirely new eventtype, whose native attributes are tailored according to the filter arguments. Additionally, the output of the **join** filter will have the same eventtype as its *primary* input streamset, and the output eventtype of the **filter-chain** filter composition tool will depend on the particular filters it aggregates. Finally, the optional `:transform` argument[3] can be used to produce an output streamset containing events of an entirely different eventtype than the input streamset.

3.3 Core Operators

This section gives a detailed explanation of the syntax and behavior of the basic FITSL primitives. Streamset splitting and elementary filtering will be described, in addition to the streamset transformations of extraction, flattening, and joining. While this chapter focuses on the independent usage of each operator, it is important to realize that these primitives are most powerful when used in combination with one another; extensive examples of such application may may be found in the next chapter.

For the rest of this chapter, syntax symbols containing the token "*ATTRIBS-SPEC*" will be used to indicate an argument used to specify attribute names, which can either

[2]Also note that the input streams to the `join` filter are likely to be of differing eventtypes.
[3]The `:transform` option is accepted by the `'test` `'xform`, `'diff`, `'diff-n`, and `'regexp` filters.

be a single symbol (e.g. `'foo`) or a list of one or more symbols (e.g. `'(foo bar baz)`).

It should also be noted that each FITSL primitive operation can be performed via a call to the **`filter`** function; some primitives may also be invoked directly, for example as can be seen in the three pairs of equivalent forms listed in Figure 3-3.

```
;; Estream-set,
;;  containing events of type:
     <eventtype 'EXAMPLE-DATA>
;; with native attribs:    (NAME PROBLEM-ID SCORE LATE-P)
((anne 1 30 T)
 (anne 2 18 NIL)
 (bart 1 23 NIL)
 (bart 2 19 NIL)
 (carl 1 24 NIL)
 (carl 2 20 T))
```

Figure 3-1: Example User Streamset

```
;; Estream-set,
;;  containing events of type:
     <eventtype 'GRADE-DATA>
;; with native attribs:    (NAME GRADE)
((anne 80.4)
 (carl 72.8))
```

Figure 3-2: Example Grade Streamset

In the following sections, examples will be given which manipulate the data shown in Figures 3-1 and 3-2.

3.3.1 Splitting Modifiers

Every streamset possesses a set of *split keys.* When a streamset S has the split-key set ψ, each split key in ψ is the name of some attribute possessed by events in S. Let the "split fingerprint" σ_ψ of some event in S be the tuple of values held by that event for the attributes named by ψ. All events in S with the same split fingerprint σ_ψ will be found in the same stream within S, and no events with differing split fingerprints will be found in the same stream. Thus, a streamset will contain as many streams as

there exist unique split fingerprints in that streamset[4], and a streamset with no split keys will contain just one stream. We say that a streamset S *is split by* attribute A if A is a member of S's split-key set.

As streamsets are immutable, the split-key set of a given streamset may not be changed. However, a transformed child streamset may be created, such that both the parent and child contain the same events, but have different split-key sets from one another.

```
(split streamset attrib-1 [attrib-2 ... ])
(filter streamset 'split attrib-1 [attrib-2 ... ])

(unsplit streamset)
(filter streamset 'unsplit)

(split-only streamset attrib-1 [attrib-2 ... ])
(filter streamset 'split-only attrib-1 [attrib-2 ... ])
```

Figure 3-3: Syntax of Split Operators

```
(progn
  (setf split-userproblem (split my-data 'name 'problem-id))
  (princ split-userproblem))

;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
     <eventtype 'EXAMPLE-DATA>
;; with native attribs:    (NAME PROBLEM-ID SCORE LATE-P)
;;  and split by: (NAME PROBLEM-ID)
((anne 2 18 NIL))
((anne 1 30 T))
((carl 2 20 T))
((carl 1 24 NIL))
((bart 2 19 NIL))
((bart 1 23 NIL))
```

Figure 3-4: Demonstration of **split**

[4] I.e., if the streamset contains no events with a given split fingerprint, there will be no stream for that fingerprint.

```
(princ (unsplit split-userproblem))

;; Estream-set,
;;  containing events of type:
     <eventtype 'EXAMPLE-DATA>
;; with native attribs:    (NAME PROBLEM-ID SCORE LATE-P)
((anne 1 30 T)
 (anne 2 18 NIL)
 (bart 1 23 NIL)
 (bart 2 19 NIL)
 (carl 1 24 NIL)
 (carl 2 20 T))
```

Figure 3-5: Demonstration of **unsplit**

```
(princ (split-only split-userproblem 'name))

;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
     <eventtype 'EXAMPLE-DATA>
;; with native attribs:    (NAME PROBLEM-ID SCORE LATE-P)
;;  and split by: (NAME)
((anne 1 30 T)
 (anne 2 18 NIL))
((carl 1 24 NIL)
 (carl 2 20 T))
((bart 1 23 NIL)
 (bart 2 19 NIL))
```

Figure 3-6: Demonstration of **split-only**

The syntaxes given in Figure 3-3 each create a child variant of ***streamset***, containing the same events as the parent, but with different split keys than its parent. The **split** operator produces a child whose split-key set is a superset of the parent's; the split-key set of the child will be the union of the specified attribute names and the split keys of the parent. A transcript demonstrating use of the **split** operator can be seen in Figure 3-4, where the resulting streamset has been split by both name and problem-id, and thus contains six streams, one for every name/problem pair present in the dataset.

The **unsplit** operator produces a child with an empty split-key set, and the

split-only operator creates a child whose split-key set contains the specified attributes (and nothing more). The usage and effects of these operators are demonstrated briefly in Figures 3-5 and 3-6; these examples reuse the `split-userproblem` streamset which was created in Figure 3-4.

3.3.2 Basic Filters

The central filtering functionality of FITSL is provided by the `'test` and `'xform` filter types, seen in Figure 3-7. These filter modes are nearly identical; the difference will be explained below. They both generate transformed copies of the input streamset; they can add annotations to the event copies, and the `'test` filter can optionally omit individual events when copying the streamset. Again, streamsets and events are immutable, and thus while one can use filters to create mutated copies of streamsets, no existing streamset object is ever modified in any way by FITSL. Similarly, while filters may generate mutated copies of the events in the input streamset, the original events are themselves unchangeable.

```
(filter streamset < 'test | 'xform > test-function
     [ :attrib ATTRIBS-SPEC ]
     < [ :annotate-as new-attrib ] | [ :transform xform-function ] >
     [ :keepnulls boolean ]
  )
```

Figure 3-7: Syntax of Basic Filters

These filters invoke ***test-function*** once for every event in ***streamset***; on each invocation, ***test-function*** will be passed either the event itself (by default), or values extracted from that event (when the optional `:attrib` argument is supplied). A basic demonstration of the `'test` filter is provided in Figure 3-8.

When the `:attrib` argument is used, the values of the given attributes will be extracted from the event, and then are passed (in the given order) to ***test-function***.

```
(progn
  (setf even-data (filter my-data 'test #'evenp
                          :attrib      'score
                          :annotate-as 'even-score
                          :keepnulls  t))
  (princ even-data))

;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
     <eventtype 'EXAMPLE-DATA>
;; with native attribs:    (NAME PROBLEM-ID SCORE LATE-P)
;; and annotated attribs:  (EVEN-SCORE)
({ (anne 1 30 T) : EVEN-SCORE=T }
 { (anne 2 18 NIL) : EVEN-SCORE=T }
 { (bart 1 23 NIL) : EVEN-SCORE=NIL }
 { (bart 2 19 NIL) : EVEN-SCORE=NIL }
 { (carl 1 24 NIL) : EVEN-SCORE=T }
 { (carl 2 20 T) : EVEN-SCORE=T })
```

Figure 3-8: Demonstration of `'test` Filter

Optional Event Removal

When using the `'xform` filter mode, events cannot be omitted from the child streamset. But when using the `'test` filter, events can be elided: when ***test-function*** returns `nil` for an event, that event is left out of the child streamset.

However, when the optional `:keepnulls` argument is non-`nil`, the `'test` mode behaves exactly like the `'xform` mode: events are never omitted from the child streamset.

Optional Annotation or Transformation

When the `:annotate-as` keyword is specified, the child events will all be given annotations named according to its ***new-attrib*** parameter. For each event, the annotation value will be the value returned from the invocation of ***test-function*** corresponding to that event. For example, if `:annotate-as` is `'green` and ***test-function*** returns the string `"yes"` for a given event, the child streamset will contain a copy of that event which has been annotated with the attribute name/value pair `'green` = `"yes"`.

To allow the user greater control over stream transformation, instead of using the default annotation mechanism, a user-specified transformation function can be used. This is requested via the `:transform` keyword, which accepts a transform function. For each event, this function will be invoked with two arguments: the event to be transformed, and the return value from that event's invocation of *test-function*. The value returned by the transform will be included in the output streamset in place of the original event.

By way of example, when one of these filters is passed the keyword arguments

```
:annotate-as 'green
```

this is equivalent to the following arguments having been given instead:

```
:transform (make-event-annotator 'green)
```

3.3.3 Extract

As will also be described in Section 4.2.1, the **extract** primitive generates a set of completely new events, which share *some* of their attributes with the events in the input *streamset*.

```
(extract streamset :attribs ATTRIBS-SPEC)
(filter streamset 'extract :attribs ATTRIBS-SPEC)
```

Figure 3-9: Syntax of Extract Operator

The `:attribs` keyword seen in Figure 3-9 specifies which parent attributes are copied into the child streamset[5]. We can see a demonstration of this keyword in Figure 3-10, where a new streamset is created, containing events of an eventtype completely different from the `'EXAMPLE-DATA` eventtype of the events in the input `even-data` streamset (originally created in Figure 3-8). In this example, the native

[5]If the parent is split by attributes which are not extracted, FITSL will issue a warning to this effect, and the child streamset will contain streams which are "split" according to attributes its events no longer possess.

```
(princ (extract even-data :attribs '(name problem-id even-score)))

;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
    <eventtype 'EXTRACT-TYPE-5>
;; with native attribs:    (NAME PROBLEM-ID EVEN-SCORE)
((anne 1 T)
 (anne 2 T)
 (bart 1 NIL)
 (bart 2 NIL)
 (carl 1 T)
 (carl 2 T))
```

Figure 3-10: Demonstration of **extract**

attributes of the new eventtype are `'name`, `'problem-id`, and `'even-score` – precisely those attributes which were extracted from the input streamset.

3.3.4 Flatten

The **flatten** operator dimensionally reduces, or "condenses" a streamset. Each stream is condensed to a single "event", such that as many condensed events are created as there are streams in the original streamset. The condensation of any given stream is achieved by passing attribute values from that stream through a "flattening" procedure. An event is then created which contains (among other things) the value produced by the flattening procedure; this flattened event is of a new type, which is related to (but not the same as) that of events in the original streamset. Thus, each stream in the original (input) streamset corresponds to a single flattened event in the output streamset, which contains some value which was derived from its parent stream. For example, one can use **flatten** to find the *per-stream* average values of a given attribute (using the library function **statistics:mean** as a flattening procedure).

Any streamset input to the **flatten** operator is first passed though a filter which insures that the streamset is split by the attributes named via the `:by` keyword described in Figure 3-11. This assures that each stream in the resulting (intermediate)

```
(flatten streamset
        [ :fn          flatten-func ]
        [ :by          by-ATTRIBS-SPEC ]
        [ :attribs     attribs-ATTRIBS-SPEC ]
        [ :flatten-to  new-attrib ]
        [ :unsplit     boolean ]
    )
(filter streamset 'flatten [ flatten-args...  ] )
```

Figure 3-11: Syntax of Flatten Operator

```
(progn
  (setf averages (flatten my-data
                          :fn #'statistics:mean
                          :by      'name
                          :attribs 'score))
  (princ averages))

;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
      <eventtype 'FLATTEN-TYPE-10>
;; with native attribs:    (NAME)
;; and annotated attribs:  (MEAN--SCORE)
;;  and split by: (NAME)
({ (anne) : MEAN--SCORE=24 })
({ (carl) : MEAN--SCORE=22 })
({ (bart) : MEAN--SCORE=21 })
```

Figure 3-12: Demonstration of **flatten**

streamset has uniform values for the attributes named in *by-ATTRIBS-SPEC*. (For example, with the example given in Figure 3-12, the `my-data` streamset is unsplit, and so an intermediate streamset is automatically created that is split by `'name`.)

Each of these intermediate streams is then individually flattened: values for the attributes named in *attribs-ATTRIBS-SPEC* are extracted from that stream's elements, and these values are passed to the flattening function *flatten-func*. For each of these streams, a single event (of an entirely new eventtype) is then created, and annotated with the flattened value corresponding to that stream.

Recall that each of this intermediate streamset's streams is homogenous for the values of the *by-ATTRIBS-SPEC* attributes. Thus, each of the events in the resulting streamset has the native attributes specified in *by-ATTRIBS-SPEC*, initialized with the value tuple inherited from its parent stream. For example, as shown in Figure 3-12, when flattening by `'name`, there are as many events created as there are unique names, each bearing the distinct name of its parent stream. Note that these events are of a new eventtype, whose native attributes (i.e. `'name`) were specified via the `:by` argument.

Naming of Flatten Annotation

If the `:flatten-to` keyword argument is given, flatten will annotate all the output events with values for the attribute named by its ***new-attrib*** parameter (which values were output by ***flatten-func***). If `:flatten-to` is not specified, an attribute name will be automatically generated from the name of the flattening function (if available) and the names of the attributes which are being flattened (i.e. those specified in ***attribs-ATTRIBS-SPEC***). For example (as demonstrated in Figure 3-12) if the attribute `'score` is being flattened using **`statistics:mean`**, the flatten annotations will have the name `'mean--score` by default.

Optional Unsplitting of Output Streamset

Each event in the output streamset will have a unique tuple of values for the attributes named in *by-ATTRIBS-SPEC*. By default, the output streamset will also be split by *by-ATTRIBS-SPEC*, and therefore will contain exactly the same number of streams as events. However, if a non-`nil` value is specified for the optional `unsplit` parameter, the output streamset will be unsplit, and all its events will be in a single stream.

3.3.5 Join

The **`join`** operator allows the cross-referencing of related streamsets. It can be useful when importing data from multiple sources, e.g. when one has imported ITS log data

and wishes to cross-reference it with students' non-ITS test scores. It can also be useful for cross-referencing **flatten** results, when flattening by the same attributes in multiple ways (e.g. as is done at the end of Section 1.3.3).

```
(join streamset
      [ :vs    vs-streamset ]
      [ :by    by-ATTRIBS-SPEC ]
      [ :with with-ATTRIBS-SPEC ]
      [ :keep-matchless-elements kme-boolean ]
      [ :allow-multiple-vs       amv-boolean ]
   )
(filter streamset 'join [ join-args...  ]  )
```

Figure 3-13: Syntax of Join Operator

```
(princ (join averages
            :vs    grades
            :by    'name
            :with 'grade
            :keep-matchless-elements t))

;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
     <eventtype 'FLATTEN-TYPE-10>
;; with native attribs:    (NAME)
;; and annotated attribs:  (MEAN--SCORE GRADE)
;;  and split by: (NAME)
({ (anne) : MEAN--SCORE=24 GRADE=80.4 })
({ (carl) : MEAN--SCORE=22 GRADE=72.8 })
({ (bart) : MEAN--SCORE=21 })
```

Figure 3-14: Demonstration of **join**

Let us define the term "event siblings" to refer to a set of events which all have the same tuple of values for certain attributes. Using the syntax shown in Figure 3-13, the **join** filter associates events from the input ***streamset*** with their "sibling" events in the ***vs-streamset***. In this case, the sibling association is based on events having the same value tuple for their ***by-ATTRIBS-SPEC*** attributes.

The output of the **join** filter, then, contains events from ***streamset*** which have

been annotated with the names and values of their siblings' *with-ATTRIBS-SPEC* attributes. (In effect, this is similar to a SQL join [10], where the *by-ATTRIBS-SPEC* attributes would be the columns named in the "ON" clause.) We can see this in the example of Figure 3-14, where the `'name` and `'mean--score` attributes of the primary input streamset (`averages`) are present in the output streamset, and the `'grade` values from the auxiliary `grades` streamset have been added as annotations (when available).

For convenience, if one passes **`join`** the magic arguments "`:with 'all`", the *streamset* will be joined with all possible attributes from *vs-streamset*, as if *with-ATTRIBS-SPEC* had specified all those attributes of *vs-streamset* which were not already among the attributes of *streamset*.

Events Lacking Siblings

If there is an "only-child" event in *streamset* (an event which has no sibling in *vs-streamset* [6]), by default that only-child event will simply be omitted from the output streamset. (This is analogous to a SQL "inner join".) However, if *kme-boolean* is non-`nil`, then the only-child event will be included in the output streamset, and will simply lack any of the *with-ATTRIBS-SPEC* annotations[7]. (This behavior is analogous to a SQL "left outer join".)

Events with Multiple Siblings

By default, if there are any events in *vs-streamset* which share the same values for the *by-ATTRIBS-SPEC* attributes, **`join`** will error (even if none of those events have siblings in *streamset*). If *amv-boolean* is non-`nil`, then **`join`** will work with such streamsets. In this case, each *streamset* event will have one output event per sibling found in the *vs-streamset*.

[6] I.e., there are no *vs-streamset* events which have the same *by-ATTRIBS-SPEC* attribute values as the only-child.

[7] Note: this is functionally equivalent to the only-child event being given `nil` values for all the *with-ATTRIBS-SPEC* attributes.

3.4 Advanced Filters

In this section, a number of more complex filters are described. Difference (first-order) filters and higher-order filters will be explained; these are similar in spirit to discrete signal-processing filters. A description is given of the regular-expression filter facility, which provides both filtering and annotation of event patterns. Finally, explanations are given of FITSL's whole-stream filters and its facility for compound filter construction.

3.4.1 Difference Filters

FITSL's `'diff` filter makes it easy to compare every event with the immediately preceding event in its stream. These comparisons can be used to selectively remove some events, and/or to annotate events with the result from some difference function.

```
(filter streamset 'diff difference-function
     [ :attrib ATTRIBS-SPEC ]
     < [ :annotate-as new-attrib ] | [ :transform xform-function ] >
     [ :keepnulls kn-boolean ]
     [ :do-leadin dl-boolean ]
 )
```

Figure 3-15: Syntax of Difference Filters

This filter works very much like the `'test` filter, except that instead of merely being passed information about the event in question, ***difference-function*** (as seen in Figure 3-15) will be also be passed information about the preceding event in the stream. (That is, when processing the nth event in a stream, ***difference-function*** will be passed information about both event n and event $(n-1)$.) A demonstration of the `'diff` filter can be found on page 57; in that example, each user's events are annotated with the time elapsed since that user's preceding event.

The `:attrib`, `:annotate-as`, `:transform`, and `:keepnulls` keyword arguments seen in Figure 3-15 have behavior identical to that described in Section 3.3.2 for the basic `'test` filter.

Events lacking Predecessors

If a non-`nil` value is specified for the `:do-leadin` operator, ***difference-function*** will be invoked once for every event. For events lacking an immediate predecessor (i.e. the first event in every stream), the value `nil` will be passed to ***difference-function*** in lieu of information about that (nonexistent) predecessor.

However, by default, the first element of every stream will be removed from the output dataset without ever invoking ***difference-function*** for that event. This allows simple difference functions[8] to be used without needing to write a wrapper function to handle the cases where `nil` would otherwise have been passed to that function.

3.4.2 Higher-order Filters

Similar to the first-order difference filter described in the preceding section, FITSL supports higher-order filters. In an nth-order filter, the filter function is given information about the current event as well as the n events which precede it.

```
(filter streamset 'diff-n filter-function
     :nth-order-diff order
     [ :attrib ATTRIBS-SPEC ]
     < [ :annotate-as new-attrib ] | [ :transform xform-function ] >
     [ :keepnulls kn-boolean ]
     [ :do-leadin dl-boolean ]
  )
```

Figure 3-16: Syntax of Higher-Order Difference Filters

As can be seen in Figure 3-16, the syntax of this filter is just like the `'diff` filter in the above section, except for the addition of the required `:nth-order-diff` keyword argument. The ***order*** parameter specified with this keyword must be a positive integer, indicating the order of the filter being constructed. The use of this filter

[8]Functions such as Lisp's built-in subtraction (`-`) or division (`/`) operators will error if one of their arguments is `nil`.

is demonstrated on page 63, where a second-order filter is used to identify episodes where users are guessing values according to a binary search heuristic.

The `'diff-n` filter operates exactly like the `'diff` filter, with two critical exceptions. First, instead of only passing ***filter-function*** information about a single predecessor event, it passes information about ***order*** predecessors. (As before, when there is no such predecessor, the value `nil` is provided in its place.) Secondly, while the difference filter defaults to a false value for the `:do-leadin` parameter, the `'diff-n` filter defaults to the value `t`. This means that, unless otherwise specified, ***filter-function*** will be invoked for all events, including the ***order*** stream elements at the beginning of each stream (which will have fewer than ***order*** predecessors).

3.4.3 Regexp Filters

To allow recognition of complex sequential patterns of interrelated events, FITSL includes a regular-expression filter mechanism. This facility finds matches for a specified pattern description in its input streamset; and can optionally remove unmatched events, and/or attach detailed annotations to its output streamset.

```
(filter streamset 'regexp regexp-pattern
     < [ :annotate-as new-attrib ] | [ :transform xform-function ] >
     [ :keepnulls boolean ]
  )
```

Figure 3-17: Syntax of Regular Expression Filters

With the syntax in Figure 3-17, ***regexp-pattern*** describes a sequence pattern which might be found in an event stream. FITSL's regexp filter scans each stream of ***streamset*** from beginning to end, attempting to make as many serial matches of ***regexp-pattern*** as possible. All events which are included in a successful match are said to "satisfy" ***regexp-pattern***. Pattern matches are not allowed to overlap, so no event can satisfy more than one match at a time. Demonstrations of the `'regexp` filter can be found on pages 17 and 63.

Non-satisfying events will be omitted from the output streamset, unless a non-`nil` value is passed via the `:keepnulls` keyword argument.

When a transformation is specified with the `:annotate-as` (or the `:transform`[9]) keyword argument, all events in the output streamset will receive the specified transformation. In the case of `:annotate-as`, each event will given an annotation named ***new-attrib*** with a boolean value (`t` or `nil`), indicating whether the event satisfied ***regexp-pattern***.

Regular Expression Grammar

The regexp parse trees used by FITSL are an extended version of that provided by CL-PPCRE [26]. While FITSL supports CL-PPCRE's full regexp syntax, a detailed explanation is beyond the scope of this document; a complete description is given in the CL-PPCRE documentation[10].

For basic regular expression composition, the only CL-PPCRE forms generally needed are `:SEQUENCE`, `:ALTERNATION`, and `REGISTER`. These are described, along with the FITSL regexp syntax extensions, in Table 3.1; for convenience, this table also provides comparison with equivalent POSIX regular expression [23] fragments. Additional demonstration of FITSL's regexp syntax can be seen on pages 17 and 63.

Annotation Nodes

The ***regexp-pattern*** may contain `:ANNOTATE` nodes, which can be used to encapsulate any regexp sub-tree. The portions of a match which are contained inside a given `:ANNOTATE` will be transformed as indicated by its ***xspec***. Much like the filter's overall `:annotate-as` and `:transform` arguments, the ***xspec*** can either be an attribute name, or a function object (embedded in the regexp tree). When an attribute name is given, all events in the output streamset will be given an annotation

[9]When a more generic transform is requested, ***xform-function*** will be passed the event and a boolean value indicating whether that event satisfied ***regexp-pattern***; the return value of ***xform-function*** will be included in the output streamset.

[10]This either can be read via the URL `http://weitz.de/cl-ppcre/#create-scanner2`, or can be found in the CL-PPCRE distribution available from [26].

FITSL Regexp Syntax	Analogous POSIX Regexp Element String
(:EVENT)	(Any single event)
(:EVENT ⟨*eventspec*⟩)	(Any single event which meets ⟨*eventspec*⟩)
(:SEQUENCE *a b c*)	*abc*
(:ALTERNATION *a b c*)	*a* | *b* | c
(:REGISTER *a*)	(*a*)
(:ANNOTATE *xspec a*)	(*a*) (matches of *a* are annotated per *xspec*)
(:MAYBE *a*)	*a*?
(:STAR *a*)	*a**
(:PLUS *a*)	*a*+
(:ATLEAST *n a*)	*a*{*n*,}
(:REPEAT *n a*)	*a*{*n*,*n*}
(:NOMORETHAN *n a*)	*a*{0,*n*}
:START	^
:END	$

Table 3.1: Overview of FITSL Regexp Syntax

with that name and a boolean value (t or nil), indicating whether they satisfied that :ANNOTATE node. Similarly, when a transform function is given, all events in the output streamset will be transformed according to whether they satisfied that :ANNOTATE node.

Also, as seen below, one can make regexp comparisons with events matched by :ANNOTATE nodes, by referencing their annotation names when composing :EVENT nodes.

Event Nodes

The :EVENT node allows testing and/or comparison of event attributes. This behavior is specified by the ⟨*eventspec*⟩ mentioned in Table 3.1, which can take either of the forms indicated in Figure 3-18, depending on whether one wishes to test more than one attribute; both of these forms can be seen in use on page 17. An :EVENT node is considered to match when all of its ⟨*attribspec*⟩ tests return non-nil values.

Here, ⟨*attribspec*⟩ is shorthand for the syntax given in Figure 3-19. As shown, any portion of ⟨*attribspec*⟩ beyond the initial *attrib-name* may be omitted, though if one omits an argument, all of the arguments which follow it must also be omitted.

⟨*attribspec*⟩

(⟨*attribspec-1*⟩) (⟨*attribspec-2*⟩) ... (⟨*attribspec-n*⟩)

Figure 3-18: Expansions of ⟨*eventspec*⟩

attrib-name test-function compare-register func-arg-1 func-arg-2 [...]

attrib-name test-function compare-register func-arg

attrib-name test-function compare-register

attrib-name test-function

attrib-name

Figure 3-19: Expansions of ⟨*attribspec*⟩

The *compare-register* field is used to indicate whether the current event should be compared with some event which has been encountered in the course of the current pattern-match attempt. When its value is `nil` (or omitted), this attribspec will examine only the current event. In the case where the value of *compare-register* is an annotation name, it refers to the `:ANNOTATE` node whose *xspec* bears that name. In particular, when an annotation name (or a positive integer[11]) is specified, the current event will be compared with the first event currently matched to the corresponding regexp node.

The value of the requested attribute *attrib-name* will be extracted from the current event (and the register-captured event, when specified). The additional arguments for *test-function* are taken to be precisely whatever *func-arg* s are specified; if these are omitted, there are no additional arguments. The attribute value(s) and

[11]In the case where *compare-register* is some integer n, it refers to the nth register-capturing node encountered in a left-to-right scan of the entire regexp tree, where both `:ANNOTATE` nodes and `:REGISTER` nodes are considered to be register-capturing. However, as it can be awkward to calculate this number (or re-calculate it when modifying a regexp), users are encouraged to use annotate-names whenever possible.

any additional function arguments will be passed to ***test-function***[12] for evaluation.

Illustration of Attribute-Specification Behavior

A few examples may help clarify the behavior of regexp-event-node attribute specifications. When an attribute specification of the form `'(ps-num = nil 3)` is testing an event belonging to problem-set number 17, the Lisp expression `(= 17 3)` will be evaluated.

Similarly, if the regexp sub-tree `'(:ATTRIB blue (:EVENT))` has been matched against an event which has the value `"yes"` for its `'answer` attribute, and the regexp engine is attempting to find a match for a node with the event-node attribute specification `'(answer` **`string=`** `blue)`. If the event currently being considered has the `'answer` attribute value `"foo"`, the regexp engine will evaluate the Lisp expression `(`**`string=`** `"foo" "yes")` in order to determine whether that attribute specification was satisfied.

Testing Regexp Filters

An helper tool has been provided to to help users debug their regular expressions. The function **`show-matches`** accepts a streamset and a regexp structure as arguments; for each stream, it prints the individual satisfying sequences which matched the regexp.

3.4.4 Whole-stream Filters

Finally, there is a whole-stream filtering mode, which allows entire streams to be examined and optionally filtered out of the resulting streamset.

```
(filter streamset 'wholestream test-function)
```

Figure 3-20: Syntax of Whole-stream Filters

[12] If *test-function* is unspecified, it defaults to the **`identity`** function, which will, in effect, test whether its argument is non-`nil`.

Using the syntax in Figure 3-20, each stream in ***streamset*** is passed (in list form) to ***test-function***. The output streamset will only include those streams for which ***test-function*** returns a non-`nil` value. A demonstration of this behavior was seen on page 19, where the `'wholestream` filter was used to produce a streamset containing only those streams from its input streamset which contained at least five events.

3.4.5 Composite Filters

Any FITSL primitive, even those like **split** and **flatten**, may be regarded as a type of filter. One may then consider the idea of cascading multiple filters to create a compound filter; such composites are here referred to as a "filter chain." When chaining filters together, the output of each filter stage is fed as input to the next stage, and the output of the final stage is returned as the filter-chain's result.

```
(filter-chain streamset
      ⟨filterspec-1⟩
      ⟨filterspec-2⟩
          ...
      ⟨filterspec-n⟩)
```

Figure 3-21: Syntax of Filter Chaining

```
(f filter-type filter-argument-1 filter-argument-2 [ ... ])
```

Figure 3-22: Expansion of ⟨*filterspec*⟩

As suggested by Figure 3-21, any number of filter-chain stages may be specified via ⟨*filterspec*⟩ statements, each of which takes the form shown in Figure 3-22. Here, ***filter-type*** is any symbol accepted as a third argument to the **filter** command (e.g. `'flatten`, `'test`, etc.), and the ***filter-argument*** parameters are the same as would follow that third argument in a non-compound filter. Use of a filter-chain was shown on page 19, where a sequence of filters was applied to achieve the desired

results. Note that in that example, the output streamset contains events of the eventtype which was created by the `'flatten` stage of that filter-chain.

Chapter 4

Other Examples

In this chapter, several examples of further use of FITSL are presented. While these hardly cover the range of experiments or applications for which FITSL may be used (as indicated in Section 1.3.4), they may yield a better understanding of the usage, operation, and power of FITSL. The first pair of examples in this section build on the scenario presented in Section 1.3, and, respectively, demonstrate some of the ways in which users can customize and extend FITSL, and explore various language features. The subsequent (and final) example uses a different dataset entirely, and illustrates use of FITSL's higher-order filtering capabilities.

4.1 Extensibility and Interoperation

While FITSL itself only provides a basic set of statistical tools, users may supplement these with their own Lisp modules, and FITSL allows arbitrarily complex application of these tools to any dataset. Furthermore, in addition to exporting data to several standard file formats which can be visualized using other tools (such as Matlab, WEKA, or a spreadsheet [17, 28]), FITSL includes mechanisms for translating datasets into standard Lisp objects for use by other Lisp-based analysis packages. These capabilities are described and demonstrated in this section.

In analyzing the results from her first experiment (in Section 1.3), Alice observes that the CETI/CETI2 students seem to manifest `reviewing-lesson` behavior much

Index	Classes
0	{CETI, CETI2}
1	{UQ, FOR-6.188}

Table 4.1: Ad-hoc Class Grouping

less often than the UQ/6.188 students. She measures this quantitatively, and finds the statistics seen in Figure 4-1. As can be seen, their interactions (on average) involve such `reviewing-lesson` behavior less than a fifth as often as the other students.

```
(princ (flatten (unsplit percentage-reviewing)
                :by 'class
                :fn #'mean-stddev
                :attribs    'percentage-true--reviewing-lesson
                :flatten-to 'sigfigs-%-reviewing))

;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
    <eventtype 'FLATTEN-TYPE-10>
;; with native attribs:    (CLASS)
;; and annotated attribs:  (SIGFIGS-%-REVIEWING)
;;  and split by: (CLASS)
({ (CETI2) : SIGFIGS-%-REVIEWING=((MEAN 0.39829728) (STDDEV 0.65366715)) })
({ (UQ) : SIGFIGS-%-REVIEWING=((MEAN 2.2601516) (STDDEV 2.0806499)) })
({ (CETI) : SIGFIGS-%-REVIEWING=((MEAN 0.30128536) (STDDEV 0.7430263)) })
({ (FOR-6.188) : SIGFIGS-%-REVIEWING=((MEAN 2.7379715) (STDDEV 1.7372252)) })

;; TOTAL runtime was <1 second
```

Figure 4-1: Exploring Per-Class Reviewing Behavior

4.1.1 Language Extensibility: User-Provided Functions

Alice wishes to measure this correlation between reviewing and class-group, and thus picks class-group indices which separate the CETI/CETI2 students from the others, as in Table 4.1. She then annotates the students' data with these indices, using the FITSL expressions given in Figure 4-2. Note that she filters the data using *her own* lambda function, compiling it prior to use with the toolkit macro **comp** (which returns a compiled version of the lambda sexp provided as its argument). The resulting

```
(setf grouped-%-reviewing
  (filter percentage-reviewing
         'xform (comp '(lambda (c)
                          (bool->int (not-member c '(:CETI :CETI2)))))
         :attrib      'class
         :annotate-as 'classgroup))

;; operation runtime was <1 second

;; TOTAL runtime was <1 second

==> #<ESTREAM-SET @ #x7da6359a>
 ... Estream-set containing 424 events of type FLATTEN-TYPE-6
 ... with native attribs:    (CLASS USERID)
 ... and annotated attribs:  (PERCENTAGE-TRUE--REVIEWING-LESSON CLASSGROUP)
 ...  and split by: (CLASS USERID)
 ...  into 424 discrete streams
```

Figure 4-2: Assigning Class-Grouping Indices

dataset, `grouped-%-reviewing`, is identical to the input (`percentage-reviewing`) dataset, but with the appropriate class-group annotation added to each event.

4.1.2 Translating Data into Standard Lisp Structures

```
(show-correlation
 (->xy (unsplit grouped-%-reviewing)
       :attribs '(classgroup percentage-true--reviewing-lesson)))

** This is a statistically significant correlation!

;; TOTAL runtime was <1 second

==> ((QUERY::R . 0.37606215) (QUERY::P . 1.0840298e-15))
```

Figure 4-3: Measuring Group/Behavior Correlation

Now Alice wishes to export values from her dataset for evaluation by a non-FITSL Lisp procedure (**`show-correlation`**). Conveniently, the FITSL toolkit procedure **`->xy`** extracts values from a streamset and stores them in a representation more usable by non-FITSL procedures. Given a streamset, it produces a normal Lisp list object, each of whose members is a list of the values extracted from one FITSL event.

Thus, as seen in Figure 4-3 she measures the statistical significance of this observed difference. This is done by using the **->xy** tool to extract the specified attributes (`classgroup` and `percentage-true--reviewing-lesson`) from the dataset events and passing the result (an ordinary list of numbers) to **show-correlation**. By yielding a correlation (r value) of 0.38, this shows that there is a definite correlation between class-group and frequency of incidence of `reviewing-lesson` behavior. Even more strikingly, Alice obtains a p-value of $1.1\text{x}10^{-15}$, which dramatically exceeds any reasonable criterion for statistical significance.

Thus, we have demonstrated how FITSL allows users to write their own extensions to the language, both internally – custom filter functions – and externally – translating datasets for use by (possibly user-written) non-FITSL Lisp functions.

4.2 Demonstration of Language Features

This section presents an arbitrary experiment that illustrates some of the other interesting features of FITSL. In this experiment, Alice explores the notional hypothesis that there is a correlation between class membership and trends in each student's rate of ITS activity. In the process of doing so, she demonstrates FITSL's **extract** primitive, its difference filters, the power of the stream-flattening facility, and its suitability for data exploration.

4.2.1 The extract Primitive

When manipulating data, one may perform annotations which are of particular interest; similarly, some of the original data fields may no longer be needed. To improve computational consumption of space and time, one might wish to create a new event-type which contains only the data attributes of interest. The **extract** primitive provides this functionality: it creates a new data-type tailored to the specified list of attributes, and creates a new streamset, whose attribute values are extracted from the original streamset.

Thus, one can create a new compact dataset from a larger dataset, trimming

```
(setf interesting-parts
  (extract (unsplit everyone-data)
           :attribs '(class userid time)))

;; operation runtime was 39 seconds

;; TOTAL runtime was 1:52 (mm:ss)

==> #<ESTREAM-SET @ #x74910a9a>
 ... Estream-set containing 334058 events of type EXTRACT-TYPE-3
 ... with native attribs:    (CLASS USERID TIME)
 ...  and split by: NIL
 ...  into 1 discrete stream (of length 334058)
```

Figure 4-4: Demonstrating **extract** Operator

unneeded fields, and rolling any desirable annotations into built-in event attributes. (Additionally, it is somewhat more efficient to manipulate built-in attributes than annotation values, due to the details of FITSL's implementation.) An example of the **extract** primitive in use can be seen in Figure 4-4. Because only the students' class, user-ID, and submission-times are of interest in this experiment, Alice begins by creating a new version of the dataset containing only these attributes.

4.2.2 Basic Difference Filters

```
(setf students-with-durations
  (filter (split interesting-parts 'class 'userid)
          'difference #'-
          :attrib      'time
          :annotate-as 'duration))

;; operation runtime was 1 second
;; operation runtime was 4 seconds

;; TOTAL runtime was 5 seconds

==> #<ESTREAM-SET @ #x78a8ff7a>
 ... Estream-set containing 333634 events of type EXTRACT-TYPE-3
 ... with native attribs:    (CLASS USERID TIME)
 ... and annotated attribs:  (DURATION)
 ...  and split by: (CLASS USERID)
 ...  into 419 discrete streams
```

Figure 4-5: Demonstrating Difference Filter

One can apply a difference filter (akin to a first-order signal processing filter), comparing each event in a stream with the event immediately preceding it. Here (as shown in Figure 4-5), Alice measures the "duration" of each event[1]: the amount of time elapsed since that student's previous event.

Each event in `interesting-parts` is annotated with the difference between its time and that of the student's previous event, as calculated using the Lisp subtraction operator (`-`). Note that, while this example is numeric, FITSL difference filters can manipulate any data type for which some kind of difference (or comparison) function can be implemented in Lisp. Also, while the above is an elementary first-order difference filter, FITSL abstracts away all the laborious work involved in applying such a filter to a streamset, presenting an interface which allows one to succinctly encode the salient features of any first-order filter.

4.2.3 The **`flatten`** Primitive

This section illustrates the expressive power provided by the combination of FITSL's streamset abstraction with its **`flatten`** primitive. As will be shown, one can use a concise **`flatten`** expression to apply any given flattening procedure to every stream in a streamset.

For the purposes of this example, we assume that trends in a student's event "durations" provide insight into that student's learning process[2]. We further assume (for the sake of example) that a rough measure of such trends can be found in the slope of the linear best-fit of each student's set of [time, duration] data-points.

Starting with `students-with-durations`, which is one-stream-per-student, Alice takes the slope of the regression line for each student's set of time/duration pairs as shown in Figure 4-6. (To streamline this example, we eliminate students with fewer than 20 interactions, and students for whom the linear-regression procedure was unable to find a fit.) Thus, Alice transforms each of `students-with-durations`'s

[1] More precisely, she measures the *upper bound* of the time the student has spent thinking about the problem since their last submission.

[2] For example, if their durations grow progressively shorter over time, one might imagine this indicates relative mastery of the material.

```
(setf time-with-linregression
  (filter-chain students-with-durations
        (f 'wholestream
           (comp '(lambda (stream) (< 20 (length stream)))))
        (f 'flatten
           :by         '(class userid)
           :attribs    '(time duration)
           :fn         #'linregress-slope
           :flatten-to 'slope)
        (f 'test #'not-null :attrib 'slope)
        (f 'unsplit)))

;; operation runtime was <1 second
;; operation runtime was 1:41 (mm:ss)
;; operation runtime was <1 second
;; operation runtime was <1 second

;; TOTAL runtime was 1:41 (mm:ss)

==> #<ESTREAM-SET @ #x80ad3752>
 ... Estream-set containing 397 events of type FLATTEN-TYPE-7
 ... with native attribs:    (CLASS USERID)
 ... and annotated attribs:  (SLOPE)
 ...  and split by: NIL
 ...  into 1 discrete stream (of length 397)
```

Figure 4-6: Applying Linear Regression to Each Stream

streams using **linregress-slope**, producing one event datum per student, containing their UID, class, and resulting regression slope.

The above demonstrates the expressive power produced by FITSL's data processing paradigm and abstractions. Using a concise **flatten** expression, the requested pair of attribute values were extracted from the 300,000 events in the streamset, and nearly 400 distinct linear regressions were performed – one for each stream in the input dataset.

4.2.4 Data Exploration with FITSL

Having calculated the desired values, Alice now compares them in a variety of ways. To get a feel for the data, she begins with some overall statistics. In Figure 4-7, we see Alice displaying the mean slope for each class, and the standard deviation of the

```
(princ (flatten time-with-linregression
                :by         'class
                :attribs    'slope
                :fn         #'mean-stddev
                :flatten-to 'slope-stats))

;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
    <eventtype 'FLATTEN-TYPE-11>
;; with native attribs:    (CLASS)
;; and annotated attribs:  (SLOPE-STATS)
;;  and split by: (CLASS)
({ (CETI2) : SLOPE-STATS=((MEAN 0.0075482796) (STDDEV 0.011864868)) })
({ (UQ) : SLOPE-STATS=((MEAN 0.013770722) (STDDEV 0.061549403)) })
({ (CETI) : SLOPE-STATS=((MEAN 0.04499874) (STDDEV 0.14899734)) })
({ (FOR-6.188) : SLOPE-STATS=((MEAN 0.0037594985) (STDDEV 0.004708432)) })

;; TOTAL runtime was <1 second
```

Figure 4-7: Examining Mean Slope, Per-Class

Index	Classes
0	{FOR-6.188, CETI2}
1	{UQ}
2	{CETI}

Table 4.2: Ad-hoc Class Indexing

slope within each class.

The interactive nature of FITSL makes it easy for Alice to further investigate this apparent difference. Noting the differences among the classes' average slopes, she decides to look for a correlation between class and average slope. She posits the three-group ordering in Table 4.2, which orders the classes by average slope, from lowest to highest. Alice then tests for a correlation between students' slopes and class-groups. However, as shown in Figure 4-8, she finds the overall correlation is relatively small, if statistically significant.

Referring back to the results in Figure 4-7, Alice sees the average CETI slope is at least three times that of the other classes. Thus, she posits a new class-grouping, separating CETI from the other classes (as in Table 4.3). Alice is able to calculate the

```
(progn
  (setf durslope-class-indicator1
    (filter time-with-linregression
            'test (make-thunk-1 #'group-index '(:UQ) '(:CETI))
            :attrib     'class
            :annotate-as 'classind))
  (princ
   (flatten (unsplit durslope-class-indicator1)
            :fn         #'show-correlation
            :attribs    '(classind slope)
            :flatten-to 'classind-slope-correlation)))

;; operation runtime was <1 second
** This is a statistically significant correlation!
;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
      <eventtype 'FLATTEN-TYPE-15>
;; with  annotated attribs:  (CLASSIND-SLOPE-CORRELATION)
({ NIL : CLASSIND-SLOPE-CORRELATION=((R . 0.122425966) (P . 0.014244958)) })

;; TOTAL runtime was <1 second
```

Figure 4-8: Experimenting with a Class Grouping

Index	Classes
0	{UQ, FOR-6.188, CETI2}
1	{CETI}

Table 4.3: Revised Ad-hoc Class Indexing

correlation among students, between their slopes and this new grouping, by making only a small revision to her expression from Figure 4-8. As can be seen in Figure 4-9, this final correlation is stronger, and more statistically significant, than what she found using the previous grouping.

Most importantly, FITSL's interactive interface makes it very natural for Alice to incrementally refine her data-analysis queries in the above manner. It is also worth noting that each of the three operations in this section took less than a second to execute. Thus we see that FITSL provides a high-level language for data exploration, and yet pays little performance penalty for this level of abstraction. Hence, FITSL

```
(progn
  (setf durslope-class-indicator2
    (filter time-with-linregression
            'test (make-thunk-1 #'group-index '(:CETI))
            :attrib      'class
            :annotate-as 'classind))
  (princ
   (flatten (unsplit durslope-class-indicator2)
             :fn          #'show-correlation
             :attribs     '(classind slope)
             :flatten-to 'classind-slope-correlation)))

;; operation runtime was <1 second
** This is a statistically significant correlation!
;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
     <eventtype 'FLATTEN-TYPE-19>
;; with  annotated attribs:  (CLASSIND-SLOPE-CORRELATION)
({ NIL : CLASSIND-SLOPE-CORRELATION=((R . 0.13550824) (P . 0.006712589)) })

;; TOTAL runtime was <1 second
```

Figure 4-9: Trying an Alternate Class Grouping

provides an efficient engine for manipulation of time-series datasets, and is therefore well-suited to interactive exploration of datasets and manual iterative refinement of query statements.

4.3 Higher-Order Filters

In addition to the first-order (difference) filters demonstrated in Section 4.2.2, FITSL can perform higher-order filtering. Here, we show how a simple second-order filter can be used to recognize when students are submitting values according to a binary search pattern. Also, this example demonstrates the analytic power which can result from the combination of multiple **flatten** operations.

Unlike previous examples, this example uses a dataset from an online analog-circuits tutor[3]. This ITS featured a circuit design problem which required students

[3]This ITS was written as part of the Cambridge-MIT Institute's Intelligent Book project [6].

to provide component values to incrementally synthesize a completed circuit meeting certain requirements. The tutor gave students detailed feedback on submitted values, such that one might expect a skilled student to reach many answers via a binary search.

4.3.1 Recognizing Binary Search

```
(setf binsearch-results
  (let ((*binsearch-fuzz* 0.17))
    (filter-chain synthesis-input-values
        (f 'split 'userid 'varname)
        (f 'diff-n #'binary-searching
           :nth-order-diff 2
           :attrib         'varval
           :annotate-as    'binsrch-tail
           :keepnulls      t)
        (f 'regexp '(:SEQUENCE
                     (:ANNOTATE start
                        (:EVENT binsrch-tail #'not))
                     (:EVENT binsrch-tail #'not)
                     (:ATLEAST 2 (:EVENT binsrch-tail)))
           :annotate-as 'binsearch
           :keepnulls   t))))

;; operation runtime was <1 second
;; operation runtime was <1 second
;; operation runtime was <1 second

;; TOTAL runtime was <1 second

==> #<ESTREAM-SET @ #x71b4bcba>
 ... Estream-set containing 1238 events of type EXTRACT-TYPE-3
 ... with native attribs:    (USERID TIME VARNAME VARVAL)
 ... and annotated attribs:  (BINSRCH-TAIL START BINSEARCH)
 ...  and split by: (USERID VARNAME)
 ...  into 176 discrete streams
```

Figure 4-10: Annotating Episodes of Binary-Searching

The log of all students' work through this problem is represented in the streamset `synthesis-input-values`. As shown in Figure 4-10, this data is then split into sepa-

The tutor was used in MIT's 6.002x class, an experimental ITS-enhanced version of the MIT course *Circuits and Electronics* (6.002).

rate streams, divided by student and circuit-variable name; it is then passed through the second-order **`binary-searching`** filter[4] for annotation of all binary-search tails (as `binsrch-tail`). Naturally, the first two events of any binary-search episode will not be tagged by this filter; they are recognized and annotated (as `binsearch`) by the regular-expression filter which follows the second-order filter.

Thus, we see how FITSL can apply any second-order filter to a dataset. Furthermore, there is no limit on the degree of filter which may be used with FITSL, as filter-order is one of the configurable parameters of the `diff-n` filter type.

4.3.2 Data Analysis with FITSL

Next, we wish to know more about how often binary search was performed. This is calculated using FITSL in Figure 4-11: first, each student's binary-search episodes are counted, and then we count how many students had any given number of binary-search episodes. The results can be seen at the bottom of Figure 4-11; of the 35 students in the class, only 13 actually used binary search to solve the problem, although one student did use it relatively heavily – nine distinct episodes.

This section merely highlights what has been shown throughout all the examples given here — that FITSL provides a high-level grammar in which one may encode arbitrary processing of multidimensional sequence data.

[4]We can characterize binary search as any stream of values where the successive *differences* between values drops by a factor of 2 with each new difference. Event number n can be said to be part of the tail of a binary search if the absolute difference in values between events n and $(n-1)$ is roughly half the absolute difference in values between events $(n-1)$ and $(n-2)$. All three, of course, make up part of the binary search episode. Thus, the tail of a binary search can be detected by a relatively simple second-order filter.

```
(let ((binsearch-count
         (filter-chain binsearch-results
            (f 'split-only 'userid)
            (f 'flatten
               :by         'userid
               :attribs    'start
               :fn         #'count-true
               :flatten-to 'binsearch-episode-count)
            (f 'split-only 'binsearch-episode-count)
            (f 'flatten
               :by         'binsearch-episode-count
               :attribs    'userid
               :fn         #'count-true
               :flatten-to 'users)
            (f 'unsplit))))
  (princ binsearch-count))

;; operation runtime was <1 second
;; operation runtime was <1 second
;; operation runtime was <1 second
;; operation runtime was <1 second
;; operation runtime was <1 second
;; Estream-set,
;;  containing events of type:
      <eventtype 'FLATTEN-TYPE-11>
;; with native attribs:    (BINSEARCH-EPISODE-COUNT)
;; and annotated attribs:  (USERS)
({ (0) : USERS=22 }
 { (1) : USERS=10 }
 { (2) : USERS=2 }
 { (9) : USERS=1 })

;; TOTAL runtime was <1 second
```

Figure 4-11: Reporting on Binary Search

Chapter 5

Implementation

The goal of FITSL was to enable efficient data processing while also providing the user with a concise high-level interface for query specification. Several design criteria were decided upon, the effects of these criteria on the choice of implementation language were considered, and a number of issues were revealed in the process of implementing these criteria; this process is discussed below.

5.1 Design Criteria

A wide range of factors influenced FITSL's initial design. This section begins by describing some broad considerations on the grammar and structure of the toolkit itself. It then discusses some important considerations regarding how FITSL might be used.

5.1.1 Overall Language Considerations

It was clear from the outset that this language should be as succinct as possible while maintaining an intelligible grammar. Users composing filter queries should be allowed to focus on their work; as much as possible, the mechanisms which implement these filters should be implicit to the language. At the same time, it was crucial that the resulting sequence-data toolkit be highly flexible – it would be impossible

to anticipate all possible filters and transformations which might be desired by a user. Thus, the language must also be extensible, and therefore capable of executing arbitrary expressions specified by the user. To complicate matters further, it was expected that these user-specified expressions would typically be applied to every event in a dataset, and thus the language would need to be able to execute them efficiently. As a result, I decided to embed FITSL in an existing language, rather than implement my own parser, compiler, or library functions.

5.1.2 Modes of Operation

Sequence data sets can be very large and complex; loading and processing them can take non-negligible amounts of time. Furthermore, development of filters (especially regexp filters) often involves a process of iterative manual refinement. If the toolkit language were only accessible via pre-scripted batch jobs, this sort of incremental data-analysis process could be extremely slow. With batch jobs, a query refinement could only be tested by re-executing the complete query script, which might take tens (or hundreds) of seconds to complete. Having an interactive interface would streamline the process of query formulation by allowing users to avoid needless re-calculation of intermediate results. Therefore, the language should have an interactive interface, but must also be scriptable for batch processing.

5.2 Choice of Implementation Language

The criteria expressed in the preceding section led in the direction of choosing a language such as Java, Lisp, Scheme, or JScheme. The MIT Scheme [13] implementation was disqualified on the basis of its memory constraints: ITS datasets can easily reach hundreds of megabytes in size, but MIT Scheme can only access 64 megabytes of memory. JScheme [4] was intriguing, but is relatively inefficient; in order to have good performance, inner loops must be implemented in Java. However, the inner loops (the heart of the filter mechanisms) would all consist primarily of user-specified code, and thus users would be forced to implement their custom query logic in Java

anyway; one might as well use regular Java instead. This left only Java and Lisp; of the two, Lisp was more attractive, with its built-in read-eval-print-loop, and also the ease with which Lisp interpreters can compile dynamically-defined functions on demand.

5.3 Various Implementation Decisions

This section discusses many of the decisions made in the process of FITSL's implementation. Perhaps the most crucial ones concerned the central approach to data manipulation used by FITSL; others involved its regexp-based filters, underlying data representation, or interactions with other tools.

5.3.1 Data Manipulation Paradigm

In a tool like FITSL, the paradigm applied to the task of data processing has profound effects on the suitability of the system for various tasks. In this case, I found it necessary to adjust my approach to both the representation and transformation of data.

Collections of Sequences

When designing FITSL, my initial ideas involved a single stream of events, plus an explicit iteration facility which could break any given stream down along one or more axis for various forms of processing (e.g. statistical analysis). However, as my conception of the language evolved, I realized it would be more desirable to shift to an approach more like my eventual streamset/splitting model. This was in part due to the conciseness achieved when iteration becomes implicit to the language grammar: primitives operate on sets of sequences, rather than individual sequences. As it was evident from the start that my ITS data would need to be broken down along many axes on a regular basis, this elegance alone was adequate reason to choose the streamset paradigm.

Flattening

Similarly, in my early attempts designing a data-analysis mechanism for FITSL, I ended up with a tool which would translate each stream into a list of values (similar to the current **->xy** tool). Then, I would process these lists with various statistical tools. This, however, decoupled the statistical results from the original dataset: stream-related information was generated which was not stored in a streamset, and which was no longer annotated with any information about the parent stream from which it had been generated. Confronted with this, I realized I was ignoring the power of my streamset abstraction: it would be much more elegant to do all FITSL processing in terms of streamsets. Hence my **flatten** primitive, which translates a streamset into a set of "events," each of which associates information about its parent stream with the flattened value derived from that parent.

The streamset paradigm I first envisioned turned out to be even more valuable than I originally expected. In part, this is due to the many different kinds of multidimensional analysis which can be performed when combining the **split** and **flatten** operators to transform sequential datasets, as has been discussed in Sections 3.3.4 and 4.2.3.

5.3.2 Regular Expression Filters

Early on, observing the descriptive power of SDL (c.f. Section 2.3) made it clear that regular expressions can provide a powerful mechanism for describing patterns in data. Thus, from the very beginning, it was planned that FITSL should include filters for which user-specified regexps determined which events to annotate.

I decided to take advantage of CL-PPCRE, an existing open-source Lisp regexp package [26]. Extensive modifications were required to allow it to manipulate lists instead of character-arrays, and to allow FITSL events as regexp atoms instead of just characters. This package package uses an S-expression (sexp) grammar to express regexps, which can be used encode any traditional POSIX [23] (or Perl [25]) character-based regexp. While CL-PPCRE includes some facilities for defining static

regexp synonyms (symbols which map to predefined regexp trees), it lacked any sort of parameterized macro facility. Thus, I implemented an extensible regexp-tree-transformation mechanism; in addition to defining a number of regexp macros (like `:STAR` and `:PLUS`), this mechanism is used to translate my `:EVENT`[1] and `:ANNOTATE` nodes into a form understood by CL-PPCRE.

My `:ANNOTATE` nodes are implemented as a variant of the `:REGISTER` nodes native to CL-PPCRE. Originally, their sole function was for specifying annotations. However, CL-PPCRE does not provide register-naming, and hence only allows reference to registers by number (according to their order of occurrence); as I experimented with my `:EVENT`-node facility for comparison with register-captured events, I saw the awkwardness of only using numbers to refer to registers. Therefore, I modified my tree-transformation modules so `:EVENT`-node comparisons could refer to registers by their `:ANNOTATE` label as well. This results in a much more elegant syntax, which greatly simplifies the process of hand-composing a regexp, particularly when iterative refinements are involved.

One may regard use of regular expressions as a powerful tool for describing patterns in sequence data, much more subtly nuanced than other forms of sequence filters. On the other hand, regexp filters can be seen as a logical extension of applying higher-order filters to a sequence. To wit, one may regard a regexp filter as a (potentially) infinite-order filter, the result of compiling the specified regular expression into a finite state machine which recognizes the described pattern.

5.3.3 Data Representation

To allow effective manipulation of large datasets, it was important to choose object representations which would allow efficient execution of typical tasks. Here, the two most frequently manipulated object types are discussed: the underlying event (datum) representation, and the objects in which stream events are sequenced.

[1]In fact, translation of `:EVENT` nodes actually involves dynamic code-generation. Because it will be executed very frequently in the course of a regexp match, the generated Lisp code is compiled at the time of tree-translation.

Event Representation

I began with the assumption that a typical data-processing session would consist of accretive annotation and filtering of a modest number of very large datasets. Based on this, I decided it was key that the underlying event-data representation avoid unnecessary duplication of data structures, and therefore reuse (i.e. share) data structures as much as possible. To facilitate this, I elected to make the event-data structures immutable, so that an annotated variant of some parent event object could safely share as much of the internal structure of its parent as possible, without worrying about the shared data being modified by subsequent operations on its parent or siblings.

To this end, I decided that an annotated (child) copy of a parent event should consist simply of the annotation (a name/value pair) plus a pointer to the original parent event. Since no data need be copied, annotating a large dataset is extremely efficient both in terms of computational space and time. By this convention, the annotation structure of any given event is effectively a linked list.

Obviously, searching such a linked list for a particular annotation is not very efficient. As an optimization, when importing a dataset into FITSL, a new data-structure type (called an *eventtype*) is defined, with fields tailored to the event-attributes contained in that dataset. Accessors for these fields (which run in constant time) are generated and cached. Each event in the dataset is translated into an *event-seed* – an appropriately initialized instance of this new structure. Like the annotation structures, this event-seed is also immutable. With this modification, an annotated event now contains a pointer to its ancestral event-seed in addition to a pointer to its parent's annotation structure.

In this scheme, attributes contained in the event-seed (which have constant-time accessors) are referred to as "native attributes." This is to distinguish them from the "annotated attributes" which are added via annotation (and whose accessors run in $O(n)$ time with the number of annotations).

Stream Representation

While our fundamental data unit is an event, FITSL is a sequence-data processing language; as such, all events are contained within streamsets. For ease of initial implementation, the streams (sequences) within these streamsets were implemented as Lisp lists, which have a linked-list structure. Most filtering operations access every stream element from first to last, and many filtering operations remove some elements, producing a shorter overall stream. Both of these characteristics make a linked list a natural and convenient choice for stream representation; the advantages to be gained from using an array instead of a linked list are questionable, and may be outweighed entirely by the increased complexity of implementation.

Indeed, it appears that the chosen data representations have given FITSL reasonable performance for annotation, splitting, and basic filtering of very large multidimensional streamsets. As can be seen in Chapter 4, any of these operations can be performed in at most a handful of seconds using a dataset containing 300,000 events, on the system used for these experiments.

5.3.4 Modular Data Import/Export Functionality

To facilitate interoperation between FITSL and other existing analysis/visualization tools, a pair of complementary frameworks for data import and export were created. Each of these frameworks allows a given file format to be encoded with a minimum of effort; simple formats like Matlab and CSV can be specified in perhaps a dozen lines of code. FITSL currently includes definitions which allow the reading and writing of files in Matlab, ARFF and CSV formats; more formats can be defined by the user. This interoperability is intended, in part, to compensate for FITSL's lack of a graphical front-end for data visualization.

Many common log formats are stateless, with one log entry per line. One can write a parser for such log-entry formats, which can then be linked into the data-import framework with a minimum of difficulty.

None of the above file formats, however, support fields that contain arbitrary data

objects (e.g. lists). Because streamsets possessing such values can be useful, it was necessary to create a FITSL-native file format; this is referred to as the `fd` format (for Fitsl Data). Any streamset can be saved in this format and later re-loaded, from another FITSL session, with exactly the same state as before.

5.4 Summary

We have thus seen how the design of FITSL strove for a compact yet comprehensible grammar, which led to an implicit abstraction for iteration over collections of sequences, as well as a succinct mechanism for analyzing such collections through use of **`flatten`** procedures. Choosing an immutable data representation, which allowed reuse of shared structure, has yielded the intended benefit of optimizing FITSL for common filtering operations. The system provides users with a high-level grammar for encoding regular expressions, including an annotation abstraction that allows comparison and annotation of events within a regexp. To ease integration with data sources and other analysis and visualization tools, the system lets users import and export information in a variety of ways, both within Lisp and via external files. Finally, the system was implemented as an extension to Lisp, providing easy access to FITSL in either batch or interpreted sessions, and giving users a highly flexible and extensible system that also allows dynamic compilation of the user-specified inner loops which can be used in FITSL's filters.

Chapter 6

Future Work

A number of potential directions for future work were made apparent in the course of implementing and using FITSL. This chapter details these: it begins by outlining various ways in which FITSL's performance might be improved, continues by discussing two minor functional enhancements which could be made, and ends by describing a few major subsystems whose addition might prove worthwhile.

6.1 Performance Improvements

While the language's core filter engine provides reasonable performance overall, there are a few operations which are not as efficient as may be desired. There are two major issues in this regard – regexp filters and unsplitting – and one minor one – the **`extract`** primitive.

6.1.1 Regular Expression Performance

Little effort has been devoted thus far to the profiling and/or optimization of FITSL's regular expression filtering module. As such, it seems likely that there are some inexpensive optimizations which could be performed. It's possible, however, that the most significant performance enhancements would be found through major rewrites of portions of the engine: either fundamental algorithmic optimizations, or a tighter

integration between the regexp engine and the data representation. The great flexibility of the modified regexp engine is both a strength and a weakness: it can encode sophisticated data patterns, but the complexity of the resulting system is liable to impede performance.

One relatively inexpensive optimization that might help dramatically would be to implement event-streams as arrays instead of linked-lists. For historical reasons, the modified CL-PPCRE engine relies heavily on naïve index-based access to sequence elements (via the Lisp **elt** operator), which means that list traversal runs in $O(n^2)$ time. It seems likely that much of the inefficiency of the regexp filter is due to such wasteful linked-list traversal; switching to an array representation for streams would allow constant-time access to sequence elements.

6.1.2 Enhancements to unsplit

Currently, it is easy to produce a transformed streamset whose split-key set is a superset of the parent streamset's, using the **split** operator. However, the only efficient mechanism for producing a transformed streamset whose split-key set is a subset of the parent's split-key set is the **unsplit** function[1]. As **unsplit** can be used only to create streamsets with empty split-key sets, there is no efficient way to *partially* unsplit[2] a streamset. This is unfortunate, because **unsplit** is a fairly expensive operation[3], and yet much of its computation may be unnecessary when only a partial unsplitting is desired. Hence, the addition of an efficient partial-unsplit facility would be a useful improvement.

If unsplitting were found to be a heavily-used operation, additional optimizations might be valuable. For example, since the overall sort order of events is generally the same in a parent streamset and all its transformed descendants, it might improve performance if this sort order were cached for use by subsequent use of **unsplit** and

[1]While **split-only** does provide this functionality, it is implemented as a wrapper which calls **unsplit** (if needed) and then **split**; as such, it typically performs much needless computation.

[2]I.e. create a streamset whose split-key set is a non-null subset of its parent's split-key set, without (as is currently done by **split-only**) first performing a complete unsplit followed by a re-splitting.

[3]Unsplitting a streamset requires merge-sorting all its streams into a single sequence.

split-only.

6.1.3 Performance of **extract**

As implemented, use of the **extract** primitive is relatively computationally intensive. While it's possible there are some optimizations which could be made to improve this situation, much of this computational expense is an unavoidable consequence of the design decisions which shaped the language. The event data structures (described in Section 5.3.3) allow very efficient execution of common operations, but such performance comes at the cost of it being comparatively expensive to create entirely new instances of such data structures (i.e. new event-seeds). Therefore, it is unclear that any significant improvement can be made to the performance of **extract** without compromising the efficiency of all other FITSL primitives.

6.2 Minor Feature Enhancements

It would be more elegant if the **join** operator allowed translation of the attribute names of its input streamsets. This could be used to avoid collision of attribute names, if both input streamsets have an attribute name in common (but which has different meanings in the respective streamsets). For example, one might wish to join a customer-data streamset and a product-data streamset, each of which had an `'id` attribute whose value were desired in the resulting joined streamset (attributes denoting, respectively, customer IDs and product IDs). The attribute in one streamset could be translated to avoid collision with the other (e.g. translating the products' `'id` attribute-name to `'product-id`). Similarly, if two datasets each used a different attribute-name to refer to the same attribute-values (e.g. `'userid` and `'uid`), one could translate them to match, so as to allow joining on this shared attribute. While such translation is not strictly necessary[4], it would be convenient to have a flexible attribute-name mapping system for the **join** operator.

[4] Using a series of simple annotations, one can achieve the same effect as such a translation.

Having a more general way to export data to non-FITSL list structures would be valuable: the `->xy` procedure described in Section 4.1.2 can process any number of attributes, but only works on streamsets which are unsplit. While the code for handling split streamsets has been written, there is no user-friendly way to invoke it or parse its results.

6.3 Major Functional Additions

The `'test`, `'diff`, and `'diff-n` filters all use the same underlying filter-processing engine. As implemented, there is no way for this engine to pass any contextual information to a (hypothetical) stateful filter-function (e.g. notification that the next event belongs to a new stream or a new streamset). This might make it difficult to implement a `'test` filter which maintained internal state, for example a filter function that internally recorded all answers submitted for a problem by a given student. To some degree, this kind of filtering can be achieved via the **`flatten`** facility, but only when the resulting dimensional reduction of the dataset is also desirable.

Also, aside from the **`flatten`** facility, there is no provision for arbitrary transformation functions that operate on the per-stream level rather than the per-event level. For example, one might imagine a machine-learning filter which must read the entirety of a stream before it can annotate any portion of the stream. While there are a number of ways this functionality might be implemented, it might be best approached as an enhancement to the `'wholestream` filter mechanism.

Finally, it would be valuable if FITSL had a stronger facility for viewing data. If nothing else, a more polished textual data display would be useful, perhaps involving a more graceful way to preview select portions of very large datasets[5]. It might also be valuable to have either some kind of basic built-in visualization framework, or to be tightly integrated with some tool which provided such functionality. On the other hand, data visualization is an area of active research providing a rich variety of tools and techniques. It may be wise for an exploration/analysis tool like FITSL simply to

[5]For example, it is nearly impossible to visually interpret a textual listing of 300,000 records.

have a flexible data-export framework (and thereby take advantage of the strengths of whatever third-party visualization tools are best suited to the task at hand) rather than investing substantial development effort in duplicating the functionality of existing tools.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Electrical Engineering and Computer Science Series. The MIT Press, second edition, 1996.

[2] Rakesh Agrawal, Giuseppe Psaila, Edward L. Wimmers, and Mohamed Zaït. Querying shapes of histories. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Twenty-first International Conference on Very Large Databases (VLDB '95)*, pages 502–514, Zurich, Switzerland, 1995. Morgan Kaufmann Publishers, Inc. San Francisco, USA.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee S. P. Chen, editors, *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press.

[4] K. Anderson, T. Hickey, and P. Norvig. Silk: a playful blend of scheme and java, 2000.

[5] Ryan Shaun Baker, Albert T. Corbett, Kenneth R Koedinger, and Angela Z. Wagner. Off-task behavior in the cognitive tutor classroom: When students "game the system". In *Proceedings of ACM CHI 2004: Computer-Human Interaction*, pages 383–390, 2004.

[6] William Billingsley, Peter Robinson, Mark Ashdown, and Chris Hanson. *Intelligent Tutoring and Supervised Problem Solving in the Browser.* 2004.

[7] Benjamin S Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16, 1984.

[8] China educational technology initiative. `http://web.mit.edu/mit-ceti/www/`, 2006.

[9] C. Collet, T. Coupaye, and T. Svensen. NAOS – Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 132–143, Santiago, Chile, 1994.

[10] C. J. Date and Hugh Darwen. *A guide to the SQL standard.* Addison-Wesley, 4th edition, November 1996.

[11] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[12] Paul Graham. *ANSI Common Lisp.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[13] Chris Hanson. MIT scheme reference manual. Technical Report AITR-1281, 1991.

[14] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD Conference*, pages 49–58, 1992.

[15] D. F. Lieuwen, N. Gehani, and R. Arlein. The Ode active database: Trigger semantics and implementation. In *Proceedings of the 12th International Conference on Data Engineering*, pages 412–421. IEEE Computer Society Press, 1996.

[16] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[17] *MATLAB User's Guide.* The MathWorks, Inc., Natick, MA 01760, 2006.

[18] Giansalvatore Mecca and Anthony J Bonner. Sequences, Datalog and transducers. In *Proceedings of the Symposium on Principles of Database Systems*, pages 23–35, 1995.

[19] MIT Department of Electrical Engineering and Computer Science. XTutor toolkit. `http://icampus.mit.edu/xtutor/`, 2005.

[20] Tadeusz Morzy and Maciej Zakrzewicz. SQL-like language for database mining. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems*, pages 311–317, St. Petersburg, 1997. University of St. Petersburg.

[21] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. In *ACM SIGMOD International Conference on Management of Data*, pages 440–451, 1997.

[22] Balaji Padmanabhan and Alexander Tuzhilin. Pattern discovery in temporal databases: a temporal logic approach. In Evangelos Simoudis, Jiawei Han, and Usama Fayyad, editors, *Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, 1996. AAAI Press.

[23] *Portable Operating System Interface (POSIX)—Part 2 (Shell and Utilities)—Section 2.8 (Regular Expression Notation).* Standard for Information Technology—Portable Operating System Interface (POSIX). IEEE Computer Society, New York, September 1992.

[24] Ido Roll, Ryan S Baker, Vincent Aleven, Bruce M McLaren, and Kenneth R Koedinger. Modeling students' metacognitive errors in two intelligent tutoring systems. In *Proceedings of International Conference on User Modeling 2005*, pages 367–376, 2005.

[25] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* O'Reilly and Associates, 3rd edition, July 2000.

[26] Edi Weitz. CL-PPCRE - portable Perl-compatible regular expressions for Common Lisp. Available via `http://weitz.de/cl-ppcre/`, December 2005.

[27] Jennifer Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD Conference*, pages 259–270, 1990.

[28] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, second edition, 2005.

[29] H A Wood and D J Wood. Help seeking, learning and contingent tutoring. *Computers and Education*, 33:153–169, 1999.