Chapter 3. Event Expressions: Filtering the Event Stream

As we discussed in Chapter 1, the Live Web is uses a data model that is the dual of the data model that has prevailed on the static Web. The static Web is built around interactive Web sites that leverage pools of static data held in databases. Interactive Web sites are built with languages like PHP and Ruby on Rails that are designed for taking user interaction and formulating SQL queries against databases. The static Web works by making dynamic queries against static data. In contrast, the Live Web is based on static queries against dynamic data.

Streams of real-time data are becoming more and more common online. Years ago, such data streams were just a trickle, but the advent of technologies like RSS and Atom as well as the appearance of services like Twitter and Facebook has turned this trickle into a raging torrent. And this is just the beginning.

The only way that we can hope to make use of all this data is if we can filter it and automate our responses to it as much as possible. As we'll discuss in detail in Chapter 4, the task is greater than mere filtering, the task is correlating these events contextually. Correlating events provides power well beyond merely using filters to tame the data torrent.

This chapter introduces in detail the pattern language, called "event expressions[1]," that we will use to match against streams of events. Event expressions are an important way that we match events with context. Event expressions are the language we use to write static queries for the dynamic data of the Live Web. Together with language structures we'll introduce later, event expressions provide SQL-like functionality for dynamic streams of data.

Patterns and Filtering

1 While the details of event expressions can be quite technical, I hope that non-technical readers will come away from this chapter with an appreciation for the kinds of patterns that can be matched and the power of using a pattern language against streams of real-time data.

When I was first introduced to the UNIX operating system in 1986 one of the most interesting and powerful commands was something called grep[2]. Grep is used to find all the lines in a file that match a user-supplied pattern. Consequently, grep is very useful for finding specific patterns of strings in files. Along with file globbing (the ability in the UNIX command line to apply a command to files that match a pattern called a "file glob"), grep allows large numbers of files to be searched for complex string patterns in a matter of seconds. Even today, I use grep almost daily.

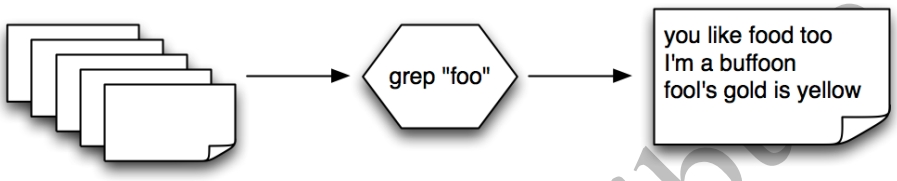

Figure 1. Using grep to select strings from a group of files that match the pattern "foo"

The simplest pattern is an exact match. The following command prints out all of the lines in the file named file.txt that contain the string "foo:"

```
grep foo file.txt
```

As shown in

Figure 1, this would match any sequence of characters that contain the subset "foo." So, the string "food" would match, as would "buffoon."

Things get more interesting when we supply more complex patterns. Patterns in grep can be full-on regular expressions, or regexs. This makes grep a very powerful tool for finding specific items. Here are a few examples of regular expressions and what they would match:

2 For a history of grep, see the Wikipedia article: http://en.wikipedia.org/wiki/Grep

- `\bfoo` – strings that contain words starting with "foo." This pattern would match "food" and "fool" but not "buffoon."
- `fo+` – strings that contain a character sequence that starts with the character "f" that is followed by one or more occurrences of the character "o." This pattern would match "food" and "fool" but not "baffled."
- `foo[^d]` – strings that contain the sequence "foo" where it is *not* followed by "d." This pattern would match "buffoon" but not "food."
- `$\d+\.\d\d` – strings that contains a dollar sign followed by any positive number of digits followed by a decimal point, followed by exactly 2 digits. This pattern would match "$15.59" and "$0.00" but not "+5.44" or "$5.666."

By using patterns we can filter large amounts of data while being very specific about what we want to find. Pattern languages, like regular expressions, provide a convenient, succinct, and unambiguous way to express complex ideas. Writing a stand-alone program to find patterns in strings that look like currency amounts (`$d+\.\d\d`) isn't rocket science, but I don't want to do it every time I need to find something in a file. With regular expressions, I express the pattern declaratively and another program translates the pattern into a program for finding the pattern.

A Pattern Language for Events

Similarly, writing a program to look at streams of events and pick out specific patterns isn't hard, but it gets tiresome when you want to do it over and over again. In the same way that we use regular expressions to find patterns in strings, we can use a declarative language to express patterns in event streams and then automatically translate those patterns into the programs that search for them.

We will call patterns in this language "event expressions" or "eventexs." As we'll see in later chapters, eventexs are used in KRL for selecting rules when a particular event pattern is present in the event stream. The processing system for KRL also uses eventexs to determine what events are salient for a particular ruleset.

An effective event expression language will have the following properties[3]:

- Power of expression
- Notational simplicity
- Precise semantics
- Scalable pattern matching

There is a trade-off between some of these properties. We can easily imagine powerful event expression languages that are not scalable or simple to write. The event expressions we describe in this chapter achieve the goal of being quite powerful while retaining their implementability, scalability, and simplicity.

Eventexs have the following benefits over natural-language descriptions and standalone programs for recognizing the same pattern:

- Eventexs save time. Eventexs are succinct. Writing an eventex is shorter than writing the equivalent program in a general purpose programming language, reducing tedium and mistakes at the same time.
- Eventexs provide a common foundation for communication. Eventexs are unambiguous. Programmers describing interesting patterns can be sure they are expressing the same concept when they use the same eventex. Similarly, programmers can be surer that the program they are writing means what they think it does.
- Event processors can translate eventexs into programs for programmatically recognizing specified patterns.
- Event generators can use eventexs to determine which events are of interest to the event processor, thereby making the event network more efficient.

We will begin by discussing the patterns that we can use to select individual events and then move on to more complex event expressions that look for patterns over streams of events[4].

Finding Individual Events

3 David Luckham, *The Power of Events*. Addison-Wesley, 2002.

4 Appendix D contains a more formal description of event expressions used in this book including the syntax and semantics of the types of eventexs we will explore in this chapter informally.

As we saw in Chapter 2, events have structure. They have a name and attributes that have values. Two of the attributes are required: a globally unique ID and a timestamp. In addition, we will allow events to include an optional domain that can be used to group events together. The particular way that this structure is expressed in event instances isn't important now so for the time being we'll ignore it[5]. Instead we will concentrate on writing patterns that can match the components of this structure. We use *primitive event expressions* for recognizing the occurrence of individual events.

Exact Matches

The simplest pattern for selecting an event from an event stream is just an event name. The following eventex selects events that have the name pageview[6]:

```
select when pageview
```

If we wanted to include a domain to ensure we were looking at events in a specific group, we could do so:

```
select when web pageview
```

This would ignore pageview events unless they also included the domain web.

While grep can be used with complex regular expressions, I suspect that in more than 90% of cases, people use it for simple, exact matches as we saw in the first example. In the same way, simple, name-only patterns may not seem powerful; however, they are among the most frequently used event expressions because, for much of what we want to do, simply matching the name is sufficient.

Attribute Matching

When exact matches aren't enough, the next step in expressive power is garnered through attribute matching. Eventexs for attribute matching are formulated in the same way as exact matching eventexs, adding as many

5 Chapter 5 will discuss the specifics of how event generators format and transmit events.
6 The keywords `select when` denote the beginning of an event expression. We'll see how this fits with KRL in Chapter 6.

attribute-regex pairs as needed to specify the precise events that should match.

The following eventex matches a pageview event and applies a regex to the event attribute `url`:

```
select when pageview url #/archives/\d{4}#
```

This pattern will match all events in the event stream that contain an attribute named `url` that has a value that contains the string `/archives/` following by four digits[7]. This particular event would indicate that the user has viewed a web page that has a URL matching the regex.

We can test more than one attribute by simply including them in the eventex. Multiple attribute-regex pairs are evaluated. All of them must be true for the event expressions to match. For example:

```
select when pageview url #/archives/\d{4}/#
                     title #iphone#i
```

This represents a subset of the events selected by the preceding eventex to only include those from the archive path that contain the word "iphone" in the title of the page. The trailing "i" in the regex indicates that case shouldn't be taken into consideration in matching the title.

Capturing Values

Regular expressions inside an eventex can be used to capture values and assign them to a variable for later use. We indicate that we want to capture a value in a regex by enclosing the part of the pattern we wish to capture in parentheses. Event expressions can use an optional setting clause to indicate the variable names for any captured values. Values are assigned to named variables in the order the captures appear in the regexs.

The following eventex would select the same events as the one in the preceding example, but also capture the digits of the archive path from the URL and the value of the word following "iphone" in the title:

---

7 The regex shown uses the hash character (#) to delimit the regular expression instead of the more common (and acceptable) slash (/) because the slash is a frequently used character in URL paths. This removes the need to quote the slash with backslashes: `/\/archives\/\d{4}\//`. Using alternate delimiters makes the regex more readable and thus communicates its meaning more clearly.

```
select when pageview url #/archives/(\d{4})/#
                      title #iphone (\w*)#i
            setting(year, next)
```

Suppose the actual event is a pageview on the path `/archives/2005/` with the page title "Singing the iPhone Blues." When the given eventex matches such an event, the variable `year` will contain the value `2005` and the variable `next` will contain the value `Blues`.

As another example, consider the following eventex that sets the variable `user_id` from the "from" address in an incoming email message:

```
select when mail received from #(.*)@windley.com#
  setting(user_id)
```

The ability to select events not just by name and domain, but also by regex matches against their individual attributes along with binding part or all of the matching values to variables, provides a powerful means of selecting events from the event stream.

When you need to use parentheses for grouping inside a regular expression but don't wish to capture the value, you can add "?:" to the front of the grouping:

```
select when pageview #/(?:archives|logs)/(\d+)/(\d+)/#
        setting (year,month)
```

The "?:" inside the first parenthesized expression keeps that match from being captured so that year and month are still set correctly. If you capture more values than you have variables in the setting clause, the extra captured values will be lost

Attribute Expressions

As powerful as regex matching is, there are times when a more freeform expression is what we need to precisely select the events in which we are interested. Instead of following the eventex name with a series of attribute-regex pairs to match attributes, the name can be followed with a single expression[8]. If the name and domain match and the expression's value is

8 Expressions will be formally introduced in Chapter 6 and are described in greater detail in Appendix E.

true, then the eventex matches. Attribute names can be used as variables in the expression.

Attribute expressions are introduced to a primitive eventex with the `where` keyword. For example, the following two eventexs mean the same thing:

```
select when pageview where url.match(#/archives/\d{4}/#)

select when pageview url #/archives/\d{4}/#
```

But suppose we only wanted to match events when the year in the archive path of the URL is greater than 2003? We could express that using regexes, but it gets messy. The following eventex accomplishes that:

```
select when pageview
            where url.extract(#/archives/(\d{4})/#) > 2003
```

The `extract` operator in this expression returns the match in the regex that is then used in the inequality test.

While only a single attribute expression can be used in a primitive eventex, we can use boolean operators to test more complex scenarios. The following eventex not only matches articles after 2003 but also requires that the title contain the string "Utah."

```
select when pageview where
        url.extract(#/archives/(\d{4})/#) > 2003 &&
        title.match("Utah")
```

Attribute expressions provide a powerful and flexible way to match individual events.

Event Scenarios

Responding to individual events is useful, but event expressions are even more powerful when used to correlate events contextually. We call a contextually meaningful, related group of events an *event scenario*. Systems that deal with event scenarios are said to do "complex event processing."

Figure 2. As you drive through town your phone notifies you that the DVD you added to your Amazon wishlist that morning is available and on sale at the Best Buy you're passing.

You can't recognize this scenarios by recognizing an event that says you added something to wishlist or an event that indicates you're driving past Best Buy. To recognize this scenario, you have to recognize the pattern of a DVD being added to the wishlist before you drive past Best Buy. Combined together, the two events provide information, automatically, that you'd otherwise have to synthesize yourself.

Complex event scenarios like this are not unusual. In fact, they make up the most interesting interactions we might have on the Live Web. We may be interested in knowing when someone is viewing a particular sequence of pages on our site or calling our support line after having sent a question by email. But creating these kinds of interactions is prohibitively expensive or even impossible using the old client-server model, with its silos of data. Complex event expressions that describe event scenarios make them easy to recognize.

Figure 2. Your DVD's at Best Buy!

Event scenarios provide much of the filtering necessary to make the potentially overwhelming streams of real-time data and events manageable. Without eventexs, the most common way to deal with complex scenarios programmatically is to build ad hoc logic that recognizes the event scenario and dispatches procedures to handle each scenario. Most modern Web applications have a chunk of logic for keeping track of the user's state in the application and then getting them to the right functionality.

But we needn't approach this important task in an ad hoc fashion. As we've seen, eventexs provide a declarative means of recognizing individual events. Our event expression language allows us to combine primitive event expressions to form compound event expressions that can be used to match the patterns of events that are common in complex event scenarios. By combining primitive events into event scenarios, developers can create sophisticated applications without the need for them to manage the state machine that would be necessary to recognize those scenarios.

Compound Events

Compound event expressions allow us to combine primitive events to form event scenarios. Event expressions provide a robust, precise notation for expressing complex patterns on event streams. Complex event expressions are not new; they have been the subject of research in database trigger languages since the 1990s and have been used since then in event engines in a variety of disciplines[9]. What *is* new is their application to real-time data streams on the Web.

Event Operators

Event operators combine event expressions into even more complex event expressions using operators that relate subexpressions to one another. Bear in mind that we're not interested in a forensic exercise where we examine logs of event occurrences. Rather, we apply event patterns to live, real-time event streams. This colors the semantics slightly.

The following binary event operators are available:

`A or B` – eventex A matches or the eventex B matches. There is no expectation of order. If either subexpression matches, then the entire expression matches.

```
select when pageview url #bar.html#
        or phone inbound_call from #801\d+#
```

In this example, the expression would match if the user viewed a page with the string "bar.html" in its URL or received a phone call from a number with area code 801.

`A and B` – eventex A matches and eventex B matches in any order.

```
select when pageview url #bar.html#
        and pageview url #foo.html#
```

In this example, the expression would match if the user viewed a page that contained the string "bar.html" in its URL and viewed another page that contained the string "foo.html." There are two events that both have to occur independently for this match to occur.

9 For a list of papers and other resources related to complex event expressions, see Appendix B.

`A before B` – eventex A matches before eventex B matches. Another way to understand this is that event A appears before event B in the event stream. The compound event matches when event B occurs. There may be intervening events between A and B.

```
select when pageview url #bar.html#
       before phone inbound_call
```

This eventex would match if the user viewed a page with the right URL before the `inbound_call` event is received. Both events have to occur before this eventex matches.

`A then B` – eventex A matches then eventex B matches with no intervening salient events.

```
select when pageview url #bar.html#
       then phone inbound_call
```

This eventex would match if the user viewed a page with the right URL and the next event signals an `inbound_call`. Both events have to occur before this eventex matches.

`A after B` – eventex matches if A occurs after B. This is equivalent to `B before A`.

```
select when pageview url #bar.html#
       after phone inbound_call
```

This eventex would match if the user viewed a page with the right URL after the `inbound_call` event is received. Both events have to occur before this eventex matches.

`A between(B, C)` – eventex A matches in between eventex B matching and eventex C matching. The compound event matches when event C occurs.

```
select when pageview url #mid.html#
              between(pageview url #first.html#,
                      pageview url #last.html#)
```

This example eventex would match if the user viewed a page with a URL that contains the string “mid.html” in between viewing pages that have URLs that contain the strings “first.html” and “last.html” respectively. Note that this eventex will match only after the pageview with “last.html” occurs.

`A not between(B, C)` – eventex A did not match in between eventex B matching and eventex C matching. The compound event matches when event C occurs.

```
select when pageview url #mid.html#
              not between(pageview url #first.html#,
                          pageview url #last.html#)
```

This example eventex would match if the user *did not view* a page with a URL that contains the string "mid.html" in between viewing pages that have URLs that contain the strings "first.html" and "last.html" respectively. Note that this eventex will match only after the pageview with "last.html" occurs.

We don't capture variables in compound eventexs. Variables can be set based on regex captures for primitive eventexs as part of the primitive event.

```
select when pageview url #mid.html#
              not between(pageview url #(\d+).html# setting(b),
                          pageview url #(\d+).html# setting(c))
```

For simplicity, the preceding examples have used a single primitive eventex (`pageview`) but there's no restriction on using different event types from different event domains in an eventex. In fact the most interesting eventexs usually involve more than one event type:

```
select when inbound_call
              from #(\d{3})\d+# setting(area_code)
    between(pageview url #custserv_page.html#,
            pageview url #homepage.html#)
```

Of course, compound eventexs can be nested. Parentheses specify order where precedent is not apparent.

```
select when pageview url #mid.html#
        between(pageview url #(\d+).html# setting(b),
                pageview url #(\d+).html# setting(c))
    before pageview url #/archives/(\d+)/foo.html# setting (year)
```

Time

Time is an important component of many events. There are three ways that we can use time: as an explicit condition on primitive eventexs, as an absolute event that is raised at an explicit point in time, and as a relative comparison of the timestamps on the components of an event expression.

Explicit Time Expressions

As we saw in Chapter 2, events contain a timestamp attribute. We can use the timestamp attribute in explicit conditions on primitive events. As the timestamp attribute is a datetime object[10], the time module operators can be used to manipulate the timestamp as part of the attribute expression of a primitive eventex.

For example, suppose your car raised an event each time it was started. We could create an eventex that only selects when the car is started before 8am as follows[11]:

```
select when car started where
   time:compare(timestamp,time:new("08:00:00")) < 0
```

Absolute Time Events

Absolute time events are similar to cronjobs in the UNIX operating system. They are not comparisons, but events that are raised at a particular time. When we create an eventex that contains an absolute time event, we are setting an alarm that will go off at that point in time. When we use them in an eventex, the eventex will not match until that alarm occurs.

Absolute time events are set using the at operator:

```
at(<datetime>)
```

This creates an alarm that will raise an event at the given datetime. The eventex matches at that specified absolute time. The parameter (`<datetime>`) is specified using datetime objects from the KRL time module of the expression language.

We could specify a page view event in between 8am and 5pm[12] as follows:

10 The built-in operator `time:new()` is used to convert strings into datetime objects.
11 The time:compare() function returns -1 if the first argument is less than the second, 0 if they are equal, and 1 if the first argument is greater than the second.

```
select when pageview
       between(at(time:new(“08:00:00”)),
               at(time:new(“17:00:00”)))
```

Note that this eventex doesn’t match when the pageview occurs, but rather when the alarm occurs at 5pm.

Here are a few other examples along with an explanation of their semantics:

```
select when at(time:new(“08:00:00”))
```

Match at 8am every morning.

```
select when at(time:new(“Friday, 07:00:00”))
```

Match at 7am every Friday.

```
select when at(time:new(“2012 May 15 07:00:00”))
```

Match at 7am on May 15, 2012.

```
select when at(time:new({day: 15}))
```

Match on the 15th of the month, every month.

We can use the before and after operators with at to specify time delineations.

```
select when pageview
       after at(time:new(“Jan 15 2011 08:00:00”))
```

When the date in a datetime string is not fully specified, we disambiguate it by assuming today. Thus the following eventex matches any pageview after 8am today:

```
select when pageview after at(time:new(“08:00:00”))
```

Relative Event Expressions

Relative event expressions compare the timestamps of the event sub expressions. The `within` operator is used:

```
A <eventop> B within n <period>
```

---

12 You might be wondering “8am in what time zone and in reference to what?” The evaluation is always referenced to the location of user for whom the event was raised.

This eventex matches only if the compound event expression `A <eventop> B` happens within the specified period. The `<eventop>` can be any of the event operators from the preceding section[13]. The `<period>` can be one of `seconds`, `minutes`, `hours`, `days`, or `weeks`. For example,

```
select when pageview url #custserv_page.html#
     before pageview url #homepage.html#
     within 3 hours
```

This eventex would match an event stream where a pageview with a URL for the customer service page came before the pageview with a URL for the homepage as long as those two events occurred within 3 hours of each other.

If the `within` clause is applied to a nested event, the period tested is between the first match and the last match of the entire nested eventex. For example,

```
select when inbound_call from #^801-\d+#
    before(pageview url #custserv_page.html# and
           pageview url #homepage.html#)
    within 3 hours
```

This eventex would match an event stream where the pageview with a URL for the customer service page and the pageview with a URL for the homepage occurred after a phone call from the "801" area code as long as the final pageview occurred within 3 hours of the inbound call.

Conditions and Alarms

Be careful not to confuse timestamp conditions with absolute time eventexs. For example, the following two eventexs are not equivalent:

```
select when car started where
   time:compare(timestamp,time:new("08:00:00")) < 0

select when car started
       before at(time:new("08:00:00"))
```

13 Note that the `within` semantics don't make sense for the `or` operator and using it in that context is not syntactically wrong, but accomplishes nothing.

The first will match as soon as the car is started, so long as the car is started before 8am. The second will match at 8am if the car was started anytime prior to 8am that same day.

Variable Arity Event Operators

The compound event operators we used previously were given as infix[14], binary operators. Event expressions also allow variable arity[15] event operators that provide more convenient methods of expressing patterns over large numbers of events. Using compound operators in this way follows this pattern:

```
select when <eventop>(E1,…En)
```

In the previous example `<eventop>` is one of `or`, `and`, `before`, `after`, or `then`. Variable arity functions are merely a convenience since their semantics can be expressed using the binary operators. For example, the following two select statements are equivalent:

```
select when and(A, B, C, D, E)

select when A and B and D and C and E
```

Note that `and` and `or` are commutative and associative, so the way that subexpressions are nested is immaterial and so the parentheses have been left out of the previous expression. The operators `before`, `after`, and `then` are neither associative nor commutative, so the nesting of subexpressions is significant. The following example illustrates the semantics of a variable arity before statement by showing the equivalent nested binary operators, properly parenthesized:

```
select when then(A, B, C, D, E)

select when A then (B then (C then (D then E)))
```

The same nesting is used for `before` and `after`.

14 The term “infix” describes operators that are placed syntactically between their operands.
15 The term “arity” refers to the number of parameters a function takes. Variable arity function can take a variable number of parameters.

Group Operators

There are three group operators `any`, `repeat` and `count`.

`any(n, E1,…,Em)` — matches if any n of eventexs E_1 through E_m match. In this eventex, n must be less than or equal to m. When n is not less than m, `any` behaves the same as a variable arity `and` operator checking for matches of all the subexpressions.

The following example shows how `any` can be used:

```
select when any(2, web pageview url #customer_support.htm#,
                   phone inbound_call to #801-649-4069#,
                   email received subject #\[help\]#)
```

This eventex would match if any 2 of the three enclosed simple eventexs matched. The preceding eventex has the same semantics as this compound eventex:

```
 select when (web pageview url #customer_support.htm# and
              phone inbound_call to #801-649-4069#)
          or
             (phone inbound_call to #801-649-4069# and
              email received subject #\[help\]#)
          or
             (web pageview url #customer_support.htm# and
              email received subject #\[help\]#)
```

I think you'll agree that the first eventex is clearer than the second.

`count(n, E)` — matches after n of eventex E have matched. Once the count eventex matches, the counter is reset and the expression begins looking for n more of eventex E.

Consider the following eventex:

```
select when count(3, E)
```

The arrows on the event stream shown in

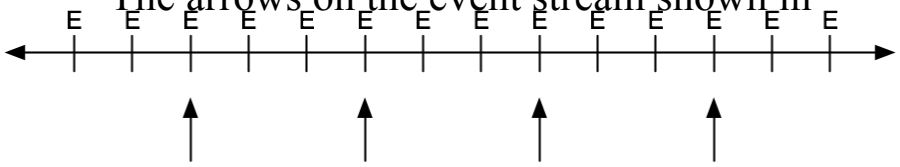

Figure 3 where this eventex would match.

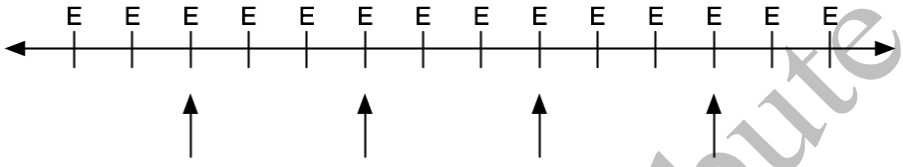

Figure 3. Event stream showing matches for `count(3, E)`

`repeat(n, E)` — matches after n of eventex E have matched. Once the repeat eventex matches, the counter is not reset and the eventex matches using a sliding window on the event stream, always matching the last n of the eventex given in the subexpression.

Consider the following eventex:

```
select when repeat(3, E)
```

The arrows on the event stream in

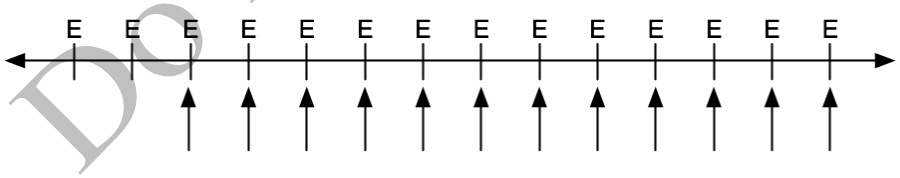

Figure 4 show where this eventex would match.

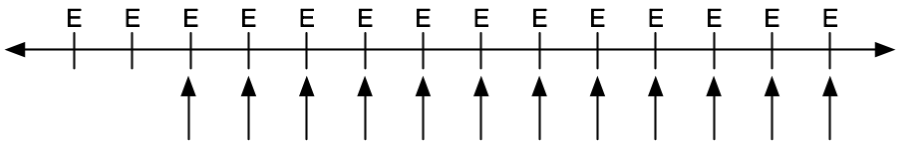

Figure 4. Event stream showing matches for `repeat(3, E)`

The repeat operator is not particularly interesting by itself, but is mostly used in convert with aggregators introduced in the next section.

We can use relative time bounds with group operators as shown in the following example:

```
select when any(2, pageview url #customer_support.html#,
                   inbound_call to #801-649-4069#,
                   email received subject #\[help\]#)
          within 3 minutes
```

This event expression has the same semantics as the any example given above except that the matches of the subexpressions must occur within 3 minutes.

To use absolute time bounds, the between operator is used:

```
select when any(2, pageview url #customer_support.html#,
                   inbound_call to #801-649-4069#,
                   email received subject #\[help\]#)
       between(at(time:new("8:00:00")),
               at(time:new("17:00:00")))
```

This eventex would match when the events matches occurred between 8am and 5pm.

Variable Correlation Between Eventexs

We've seen that the setting clause can be used to capture values in regular expressions and bind them to variables in primitive events. Once a value is bound to a variable, it can be used in subsequent subexpressions in the same eventex[16]. For example, suppose that you want to see if someone has viewed the same page twice without having to specify the page. We don't care which page they view, as long as the view it twice.

```
select when pageview url #/archives/(.*)#) setting (a)
```

16 To be less ambiguous, variables bound in one eventex can be used in enclosing eventexs. Parentheses can be used to ensure that the nesting of event subexpressions is what the developer desires.

```
before pageview url.match(("#/archives/"+a+"#").as("regex"))
```

This eventex takes advantage of the expression language's type coercion operator, as, to construct a string using the value from the first eventex and then turn it into a regex for the second match.

Aggregators

For the `count` and `repeat` operators, we sometimes want to aggregate values from the enclosed subexpression. The following clauses can be used to accumulate values:

`E max(<var>)` — accumulate the maximum value of the captured value in the variable `<var>`. The following eventex will match using a sliding window of five withdrawal events and set the variable m to have the maximum amount:

```
select when repeat(5, withdrawal amount #$(\d+\.\d\d)#) max(m)
```

As we saw above, the repeat operator will continue to match for every withdrawal event after the first five. The variable m will always contain the maximum value of the last five withdrawal events.

`E min(<var>)` — accumulate the minimum value of the captured value in the variable `<var>`. The following eventex will match five withdrawal events using a sliding window and set the variable m to have the minimum amount:

```
select when repeat(5,withdrawal amount #$(\d+\.\d\d)#) min(m)
```

`E sum(<var>)` — accumulate the sum of the captured values in the variable `<var>`. The following eventex will match five withdrawal events and set the variable m to have the sum of the amounts:

```
select when count(5, withdrawal amount #$(\d+\.\d\d)#) sum(m)
```

Because we used count rather than repeat, the eventex will sum the amounts of every five withdrawals, rather than the last five.

`E avg(<var>)` — accumulate the average of the captured values in the variable `<var>`. The following eventex will match five withdrawal events and set the variable m to have the average of the amounts:

```
select when repeat(5, withdrawal amount #$(\d+\.\d\d)#) avg(m)
```


`E push(<var>)` — append the captured values to the variable `<var>`. The following eventex will match five withdrawal events and set the variable m an array of the amounts:

```
select when repeat(5, withdrawal amount #$(\d+\.\d\d)#) push(m)
```

If the eventex that the aggregator is attached to captures more than one variable, the first variable captured is used in the aggregator. Only one aggregator can be used in an event expression.

Eventex Examples

The following examples give a scenario and a sample eventex that might be used to recognize that scenario. All of these assume the presence of endpoints that are able to recognize events of interest and are properly configured.

Large withdrawals—this scenario is fairly common and a feature built into many banking sites. The eventex selects when there is a withdrawal event where the parameter amount is over a certain limit.

```
select when bank withdrawal where amount > 100
```

Too many withdrawals—we may be interested to know when the number of withdrawals from an account passes a certain threshold during the business day:

```
select when count(4, bank withdrawal)
      between(at(time:new(“08:00:00”)),
              at(time:new(“17:00:00”)))
```

Too many withdrawals in 24 hours—rather than focusing on the business day, which might be too specific for a world of ATMs, we can use a relative time expression to match when there are 4 withdrawals in a 24-hour period:

```
select when count(4, bank withdrawal) within 24 hours
```

Too many withdrawals over a limit—we can add a limit to only match a specific count of withdrawals that are over a threshold ($100 in this case):

```
select when count(4, bank withdrawal amount > 100) within 24 hours
```


Withdrawal after a deposit—a withdrawal following a deposit matches when the withdrawal amount is greater than the deposit:

```
select when bank deposit amount #(\d+)# setting(dep_amt)
     before bank withdrawal where amount > dep_amt
```

Withdrawal after a deposit with a limit—A withdrawal following a deposit matches when the withdrawal amount is greater than the deposit or greater than a threshold:

```
select when bank deposit amount #(\d+)# setting(dep_amt)
    before bank withdrawal where amount > dep_amt || amount > 100
```

Phone call with a followup SMS—we are interested in knowing when a phone call is received within 1 hour of an SMS being received from the same number:

```
select when inbound_call from #(.*)# setting (num)
     after sms_received where from.match("/#{num}/".as("regexp"))
   within 1 hour
```

Too many phone calls—match when there are more than a threshold amount of phone calls in a given time period:

```
select when repeat(5, inbound_call) within 20 minutes
```

Too many phone calls from one number—match when there are more than a threshold amount of phone calls from the same number[17] in a given time period:

```
select when repeat(5, inbound_call from #.*#) push(nums)
       within 20 minutes
```

Looking at travel sites—match pageview events that appear to be focusing on travel related sites:

```
select when any(2, pageview url #orbitz#,
                          pageview url #kayak#,
                          pageview url #priceline#,
                          pageview url #travelocity#,
                          pageview url #expedia#)
```

17 We don't actually check that the numbers are the same in the eventex, we merely push them onto an array. A condition in the rule associated with this eventex can check to ensure they're the same. See Chapter 6 for more information on rules.

Looking for support—match when the user calls the support number within 1 day of visiting the support website:

```
select when inbound_call from app:support_number
        and pageview where url.match(app:support_website)
       within 1 day
```

Note that this example uses application variables for the support number and website regular expressions. The use of the `and` operator means that either could happen first.

Find news articles that affect stock price—when an RSS feed contains a story that includes a stock ticker symbol and the price of that same stock goes up by more than 2% within 10 minutes.

```
select when rss item content #Stock Symbol: (\w+)#
              setting (symbol)
    before stocks where price_rise ticker eq symbol && percent > 2
  within 10 minutes
```

Matching Scenarios

The examples from the preceding section show some of the power of using eventexs to match complex event scenarios. Eventexs make expressing the desired scenarios succinct and unambiguous.

The use of eventexs allows us to create dynamic queries that can filter the stream of real-time event data matching only the event scenarios that are relevant to the task at hand. Being able to look for relevant event patterns is the first step toward managing and harnessing a deluge of real-time events. In coming chapters we'll discover the power of not only matching event scenarios, but also processing them to achieve the user's purpose.

