Description of the Odin Event Extraction Framework and Rule Language

Marco A. Valenzuela-Escárcega Gus Hahn-Powell Mihai Surdeanu
Computational Language Understanding (CLU) Lab
University of Arizona, Tucson, AZ, USA
{marcov,hahnpowell,msurdeanu}@email.arizona.edu

Last Revised: September 24, 2015
Version 1.0 (see Changes)

Abstract

This document describes the Odin framework, which is a domain-independent platform for developing rule-based event extraction models. Odin aims to be powerful (the rule language allows the modeling of complex syntactic structures) and robust (to recover from syntactic parsing errors, syntactic patterns can be freely mixed with surface, token-based patterns), while remaining simple (some domain grammars can be up and running in minutes), and fast (Odin processes over 100 sentences/second in a real-world domain with over 200 rules). Here we include a thorough definition of the Odin rule language, together with a description of the Odin API in the Scala language, which allows one to apply these rules to arbitrary texts.

Contents

1 Changes

1.0: Initial release.

2 Introduction

Rule-based information extraction (IE) has long enjoyed wide adoption throughout industry, though it has remained largely ignored in academia, in favor of machine learning (ML) methods [Chiticariu et al., 2013]. However, rule-based systems have several advantages over pure ML systems, including: (a) the rules are interpretable and thus suitable for rapid development and domain transfer; and (b) humans and machines can contribute to the same model. Why then have such systems failed to hold the attention of the academic community? One argument raised by Chiticariu et al. is that, despite notable efforts [Appelt and Onyshkevych, 1998, Levy and Andrew, 2006, Hunter et al., 2008, Cunningham et al., 2011, Chang and Manning, 2014], there is not a standard language for this task, or a "standard way to express rules", which raises the entry cost for new rule-based systems.

Odin (Open Domain INformer) aims to address this issue with a novel event extraction (EE) language and framework. The design of Odin followed the simplicity principles promoted by other natural language processing toolkits, such as Stanford's CoreNLP, which aim to "avoid over-design", "do one thing well", and have a user "up and running in ten minutes or less" [Manning et al., 2014]. For example, consider a domain that tracks people's movement, as reported in the news. One may want to quickly write a domain grammar that captures events with the following arguments: (a) the subject of the verb "move" (and its synonyms) only if it has been identified as a PERSON by a named entity recognizer (NER), (b) the indirect object of the same verb that is dominated by the preposition "from" as the origin *location*, and (c) an indirect object dominated by the preposition "to" as the *destination*. Odin captures such event patterns (and more) using a single declarative rule.

In particular, Odin is:

Simple: Taking advantage of a syntactic dependency (SD) representation [de Marneffe and Manning, 2008], our EE language has a simple, declarative syntax for the extraction of *n*-ary events, which captures single or multi-word event predicates with lexical and morphological constraints, and event arguments with (generally) simple syntactic patterns and semantic constraints.

Powerful: Despite its simplicity, our EE framework can capture complex constructs when necessary, such as: (a) recursive events[1], and (b) complex regular expressions over syntactic patterns for event arguments. Inspired by Stanford's Semgrex[2], we have extended a standard regular expression language to describe patterns over directed graphs[3], e.g., we introduce new $<$ and $>$ operators to specify the direction of edge traversal in the dependency graph. Finally, we allow for (c) optional arguments and multiple arguments with the same name.

Robust: To recover from unavoidable syntactic errors, SD patterns (such as the ones shown in the next section) can be can be freely mixed with token-based surface patterns, using a language inspired by the Allen Institute of Artificial Intelligence's Tagger[4]. These patterns match against information extracted in our text processing pipeline[5] , namely a token's part of speech, lemmatized form, named entity label, and the immediate incoming and outgoing edges in the SD graph.

Fast: Our EE runtime is fast because the Odin runtime uses event trigger phrases (e.g., "move" for a moving event), which are captured with lexico-morphological patterns, as shallow filters to reduce the search space for pattern matching. That is, only when event triggers are detected is the matching of more complex syntactic patterns for arguments attempted. This guarantees quick executions. For

[1] Events that take other events as arguments. See the walkthrough example in the next section.

[2] `nlp.stanford.edu/software/tregex.shtml`

[3] We currently use Stanford syntactic dependencies, but any other graph derived from text could be used. For example, one could use a graph that models semantic roles or abstract meaning representation.

[4] `https://github.com/allenai/taggers`

[5] `https://github.com/sistanlp/processors`

The phosphorylation of **MEK** by **RAS** inhibits the ubiquitination of **TGF**.

Figure 1: A sentence containing three events in the biomedical domain: a phosphorylation, ubiquitination, and a negative regulation between the two. The text in bold marks biochemical named entities previously identified by a NER.

example, in a real-world biochemical domain, Odin processes an average of 110 sentences/second[6] with a grammar of 211 rules on a laptop with an i7 CPU and 16GB of RAM.

This document is organized as follows. Section 3 introduces the Odin rule language with a simple walkthrough example. Section 4 describes the complete rule language. The remaining sections introduce the programmatic aspects of Odin. Section 5 describes Odin mentions, which are Scala[7] objects that store the output of rules. Section 6 describes programmatic ways to customize the output of rules, by attaching custom Scala code to rules. An important note here is that Odin constructs these mentions automatically, so adding custom actions is completely optional and their addition should be reserved for complex phenomena that are not easily implemented with rules, e.g., coreference resolution. Lastly, Section 7 puts it all together, by introducing the Odin Scala API, i.e., how to instantiate and execute a domain grammar programatically.

3 A Walkthrough Example

Lets use the sentence in Figure 1 as a simple walkthrough example for an Odin grammar in the biomedical domain. This particular sentence contains three protein named entities, previously found by a NER[8], and we would like to build a grammar that finds three molecular events: two simple events that operate directly on the entities mentioned in the text, that is the phosphorylation of MEK by RAS, and the ubiquitination of TGF[9].

Conceptually, Odin follows the same strategy introduced by FASTUS more than 20 years ago Appelt et al. [1993]: it applies a cascade of grammars, where each grammar builds on the output produced by the previous one. This is illustrated in the grammar listed in Example 1, which lists all the rules necessary to capture the events of interest from Figure 1. The different rules capture multiple phenomena:

ner promotes the output of the external NER, i.e., the NE labels in IOB notation[10], to Odin's mention objects, and assigns them the arbitrary label `Protein`. Note that mention labels are a domain-dependent choice, and, thus, they are the responsibility of the domain developer. The implement this rule we used a simple surface, or `token`, pattern.

phospho matches a phosphorylation event, which is anchored around a nominal `trigger`, "phophorylation", and has two arguments: a mandatory `theme`, which is syntactically attached to the trigger verb through the preposition "of", and an optional (note the `?` character) `cause`, attached to the trigger through the preposition "by". Both arguments must be `Protein` mention. In general, we call events that take only entity mentions as arguments *simple* events. The resulting event mention is assigned the `Phosphorylation` and `Event` labels (any number

[6] After the initial text processing pipeline that includes syntactic parsing.

[7] Odin is implemented in the Scala language. However, because Scala runs on the standard Java Virtual Machine (JVM), it plays well with other JVM languages, most notably Java.

[8] Although here we focus on event extraction, Odin can also be used to write rules that extract entities. We largely ignore these type of rules here because event extraction is much more challenging and exciting.

[9] It is not extremely important in this context, but, in the biomedical domain, a phosphorylation event adds a phosphate group to the corresponding protein, which alters the activity of the protein. Similarly, ubiquitination adds ubiquitin, a regulatory protein, to the corresponding substrate protein. Finally we have a more complex event that takes these two events as arguments (phosphorylation inhibits ubiquitination). Detecting and linking these kinds of interactions, or"events", deepens our understanding of cancer signaling pathways.

[10] The IOB or BIO notation is a common representation, first proposed in Ramshaw and Marcus [1995], used to capture sequences of words that form named entity mentions. Please see `http://www.cnts.ua.ac.be/conll2003/ner/` for more examples and details.

```
- name: ner
  label: Protein
  type: token
  pattern: |
    [entity="B-Protein"][entity="I-Protein"]*

- name: phospho
  label: [Phosphorylation, Event]
  pattern: |
    trigger = phosphorylation
    theme: Protein = prep_of
    cause: Protein? = prep_by

- name: ubiq
  label: [Ubiquitination, Event]
  pattern: |
    trigger = ubiquitination
    theme: Protein = prep_of
    cause: Protein? = prep_by

- name: negreg
  label: Negative_regulation
  pattern: |
    trigger = [lemma=inhibit & tag=/^V/]
    theme: Event = dobj
    cause: Event = nsubj
```

Example 1: Rules that capture the events listed in Figure 1.

of labels ≥ 1 can be assigned through a rule). By assigning multiple labels to a mention, a domain developer essentially implements a de facto domain taxonomy. For example, in this example, we arbitrarily decide that an `IS-A` relation exists between labels from left to right. That is, the `Phosphorylation` event is a type of `Event`. In Section 4.5.2 we discuss how to use formally-defined taxonomies in Odin.

ubiq matches another simple event, this time around a ubiquitination. Clearly, there is a lot of redundancy between these last two rules. We will discuss later how to avoid this through rule templates.

negreg matches the specified trigger for a negative regulation, and then uses syntactic patterns to find the arguments, theme and cause, which, this time, must be event mentions. This rule will of course match only after the mentions for the simple events introduced above are constructed. We call these type of events, which take other events as arguments, *recursive* events.

Explicit priorities can be assigned to rules to control the order and extent of their execution. It is important to note that these priorities are not mandatory. If they are not specified, Odin attempts to match all rules, which imposes an implicit execution. That is, `phospho` and `ubiq` can only match after `ner` is executed, because they require entity mentions as arguments. Similarly, `negreg` matches only after the simple event mentions are constructed.

Once the domain grammar is defined, the hard work is done. These rules are fed into an `ExtractorEngine` Scala object that applies them on free text and returns the extracted Mentions, as summarized in Example 2.

Of course, this simple example does not cover all of Odin's features. In the following sections you will learn the different features that can be used to make more general, permissive, or restrictive rules using our declarative language. For advanced users, we will also demonstrate how to write custom code that can be attached to rules, also known as "actions", which can be used to transform the extracted mentions in ways that are not supported by the language, so that you can create complex systems that better adapt to your needs.

```
val rules = "... text containing a domain grammar ..."
// this engine applies the rules on free text and constructs output mentions
val ee = ExtractorEngine(rules)
// instantiate a Processor, for named entity recognition and syntactic analysis
val proc = new BioNLPProcessor
// annotate text, producing a document with POS, NER, and syntactic annotations
val text = "... example text ..."
val doc = proc.annotate(text)
// and, lastly, apply the domain grammar on this document
val mentions = ee.extractFrom(doc)
```

Example 2: A simple Scala API example. Here we used `BioNLPProcessor`, a processor tuned for texts in the biomedical domain, for POS, NER, and syntactic analysis. We offer open-domain processors as well, such as `CoreNLPProcessor`.

4 Rules

As the previous example illustrated, the fundamental building block of an Odin grammar is a rule. Rules define either surface patterns, which are flat patterns over sequences of words, such as **ner** in the example (formally defined in Section 4.3), or patterns over the underlying syntactic structure of a sentence described using relational dependencies, such as **phospho**, **ubiq**, or **negreg** (defined in Section 4.4).

All Odin rules are written in YAML Ben-Kiki et al. [2005]. However, it is not necessary to be a YAML expert to use Odin, as we only use a small and simple YAML subset to write rules. A brief explanation of the required YAML features is given in Section 4.1.

Once you are comfortable writing rules, it is time to construct a complete domain grammar. In the simplest instance, a complete grammar is a single file containing some rules (similar to Example 1). While this is sufficient for simple domains, when tackling more complex domains it may become necessary to organize rules into several files and recycle sets of prototypical rules to cover related events by altering sub-pattern variables. We describe all these situations in Section 4.5.

4.1 A Gentle Introduction to YAML

Odin rules are written using a small YAML Ben-Kiki et al. [2005] subset. In particular, we only use lists, associative arrays, and strings, which are briefly summarized below. For more details (although you should not need them), please read the YAML manual Ben-Kiki et al. [2005].

4.1.1 YAML Lists

YAML supports two different ways of specifying lists. The recommended one for Odin requires each list item to appear in a line by itself, and it is denoted by prepending a dash and a space before the actual element. Elements of the same list must have the same level of indentation. As an example, a list of fruits in YAML notation is provided in Example 3.

```
- apple
- banana
- orange
- watermelon
```

Example 3: Example YAML list

4.1.2 YAML Associative Arrays

YAML supports two different syntaxes for associative arrays. The recommended one for Odin is the one in which each key-value pair appears in its own line, and all key-value pairs have the same level

of indentation. Each key must be followed by colon. An example of a YAML associative array is provided in Example 4.

```
first_name: Homer
last_name: Simpson
address: 742 Evergreen Terrace
town: Springfield
```

Example 4: Example YAML associative array

4.1.3 YAML Strings

Many rule components are encoded using single-line strings, as we have seen in the previous examples. There is one exception: the rule's `pattern` field (as described in Sections 4.3 and 4.4). Patterns can be complex and it is a good idea to break them into several lines. YAML supports multi-line strings using the vertical bar character (e.g. `|`) to partition a key-value pair. When this is used, the string begins in the next line and it is delimited by its indentation. An example of a YAML multi-line string is shown in Example 5.

```
var1: single-line string
var2: |
  this is a multi-line string
  this is still part of the same string
  because of its indentation
var3: another single-line string
```

Example 5: Example YAML associative array with one multi-line string value

As shown, YAML strings don't have to be quoted. This is a nice feature that allows one to write shorter and cleaner rules. However, there is one exception that you should be aware of: strings that start with a YAML indicator character must be quoted. Indicator characters have special semantics and must be quoted if they should be interpreted as part of a string. These are all the valid YAML indicator characters:

```
- ? : , [ ] { } # & * ! | > ' " % @ `
```

As you can probably tell, these are not characters that occur frequently in practice. Usually names and labels are composed of alphanumeric characters and the occasional underscore, so, most of the time, you can get away without quoting strings.

4.2 Rules

Odin rules are represented simply as YAML associative arrays, using the fields shown in Table 1.

Field	Description	Default
`name`	The rule's name (must be unique)	*must be provided*
`label`	The label or list of labels to assign to the mentions found by this rule	*must be provided*
`priority`	The iterations in which this rule should be applied. Note that the Odin runtime system continuously applies the given grammar on a given sentence until no new rule matches (this allows grammars that use recursive events, such as the one in Example 1 to work). Each of these distinct runs is called an "iteration", and they are all numbered starting from 1. Through priorities, a developer can specify in which iteration(s) the corresponding rule should run. Specifying priorities is not required, but it may have an impact on run time, by optimizing which rule should be applied when. A priority can be exact (denoted by a single number), a range (two numbers separated by a dash), an infinite range (a number followed by a plus +), or a list of priorities (a comma separated list of numbers surrounded by square brackets.	`1+`
`action`	The custom code (or "action") to call for the matched mentions. As discussed in Section 6, specifying an action is not required. The `default` action does the most widely used job, i.e., keeping track of what was matched.	`default`
`keep`	Include the output of this rule in the output results?	`true`
`type`	What type of rule is this: surface rule (`token`) or syntax-based (`dependency`)?	`dependency`
`unit`	As discussed in Section 4.3, each token contains multiple pieces of information, e.g., the actual word (`word`), its lemma, or its part-of-speech (POS) tag (`tag`). This parameter indicates which of these fields to be matched against implicitly, i.e., when the token pattern is a simple string. Currently, the only valid values are `word` and `tag`.	`word`
`pattern`	Either a token or a dependency pattern, as specified in `type`, that describes how to match mentions.	*must be provided*

Table 1: An overview of the fields of Odin rules.

Clearly, the most important part of a rule, is the `pattern` field. In Section 4.3 we describe how to implement surface, or "token", patterns. These are useful for simple sequences, or when syntax is not to be trusted. In Section 4.4 we introduce the bread-and-butter of Odin: syntactic, or "dependency", patterns. Note that both types of patterns use some of the same constructs: string matchers (i.e., objects that can match a string), and token constraints (i.e., objects that impose complex conditions on individual tokens to be matched). We will introduce these for token patterns, and reuse them for dependency patterns.

4.3 Token Patterns

A common task in information extraction is extracting structured information from text. Structured information may refer to different kinds of things, from item enumerations to complex event mentions. One way to extract this kind of mentions from text is by the use of surface patterns that allow us to match sequences of tokens that usually signal the presence of the information we are interested in.

Surface patterns are available in Odin through the use of "token" patterns. Odin's token patterns can match continuous and discontinuous token sequences by applying linguistic constraints on each

token (Section 4.3.1), imposing structure (Section 4.3.5), generalized through the use of operators (Section 4.3.6), and drawing on context (Section 4.3.7). In this section we will describe each of these features that make token patterns efficient and easy to use for the different information extraction tasks that are encountered by practitioners.

4.3.1 Token Constraints

Remember that, in the simplest case, a token (or word) can be matched in Odin simply by specifying a string. For example, to match the phosphorylation trigger in Example 1, all we had to do was write `phosphorylation` (quotes are optional). But, of course, Odin can do a lot more when matching individual words. This is where token constraints become useful. A token constraint is a boolean expression surrounded by square brackets that can be used to impose more complex conditions when matching a token.

Each token has multiple fields that can be matched:

Field	Description
`word`	The actual token.
`lemma`	The lemma form of the token
`tag`	The part-of-speech (PoS) tag assigned to the token
`incoming`	Incoming relations from the dependency graph for the token
`outgoing`	Outgoing relations from the dependency graph for the token
`chunk`	The shallow constituent type (ex. `NP, VP`) immediately containing the token
`entity`	The NER label of the token
`mention`	The label of any `Mention`(s) (i.e., rule output) that contains the token.

Table 2: An overview of the attributes that may be specified in a token constraint.

A token field is matched by writing the field name, followed by the equals character and a string matcher. (e.g. `word=dog` matches the word "dog", `tag=/^V/` matches any token with a part-of-speech that starts with "V", `entity="B-Person"` matches any token that is the beginning of a person named entity). Expressions can be combined using the common boolean operators: and `&`, or `|`, not `!`. Parentheses are also available for grouping the boolean expressions.

Note: if the square brackets that delimit the token constraint are left empty, i.e., `[]`, the expression will match any token.

4.3.2 String Matchers

A string matcher is an object that matches a string. Matching strings is the most common operation in Odin, being heavily used both in token and dependency patterns. This is because all token fields (described in Table 2) have string values that are matched using string matchers. Additionally, dependency patterns (described in Section 4.4) match incoming and outgoing dependencies by matching the name of the dependency using the same string matchers.

Strings can be matched exactly or using regular expressions. Both options are described next.

4.3.3 Exact String Matchers

An exact string matcher is denoted using a string literal, which is a single- or double-quote delimited string. The escape character is the backslash (e.g., `\`). If the string is a valid Java identifier, the quotes can be omitted. For example, `word=dog` matches the word "dog".

4.3.4 Regex String Matchers

A regex string matcher is denoted by a slash delimited Java regular expression.[11] A slash can be escaped using a backslash. This is the only escaping done by Odin to regular expressions, everything else is handled by the Java regular expression engine. For example, `tag=/^V/` matches any token with a part-of-speech that starts with "V".

4.3.5 Named Arguments

Token patterns support two types of named arguments: those constructed "on-the-fly" from an arbitrary sequence of tokens or those that point to existing mentions.

Capturing a sequence of tokens and assigning a label to the span for later use can be performed using the `(?<identifier> pattern)` notation, where `identifier` is the argument name and `pattern` is the token pattern whose result should be captured and associated with the argument name. Capturing several sequences or mentions with the same name is supported as well as nested captures (i.e., arguments defined inside other arguments).

Bonnie and **Clyde** robbed the bank.

```
1       (?<robber> Bonnie) and (?<robber> Clyde) robbed []*? (?<location> bank)
```

Example 6: An example of a token pattern with a repeated argument using a subpattern-style named argument.

While powerful, these subpattern-style named arguments can quickly clutter a rule, especially when the pattern is nontrivial. Consider the `(?<robber>)` pattern in Example 6. A broad-coverage rule for detecting a *robber* could be quite complex. A better strategy might be to generalize this pattern as a rule designed to identify any *person*. Since this rule provides the context of a *robbery* event, it would be sufficient to simply specify that the span of text being labelled *robber* is a mention of a *person*. We can do this quite easily with Odin.

A previously matched mention can be included in a token pattern using the `@` operator followed by a `StringMatcher` that should match a mention label. This will consume all the tokens that are part of the matched mention. If the mention should be captured in one of the named groups then the notation is `@identifier:StringMatcher` where the identifier is the group name and the string matcher should match the mention label.

Bonnie and **Clyde** robbed the bank.

```
1       @robber:Person and @robber:Person robbed []*? @location:Location
```

Example 7: An example of a token pattern with a repeated argument using an mention-based named argument. This assumes that other rules built the `Person` and `Location` mentions, possibly from the output of a NER.

4.3.6 Token Pattern Operations

The most fundamental token pattern operations are concatenation and alternation. Concatenating two patterns is achieved by writing one pattern after the other. Alternation is achieved by separating the two patterns using the alternation operator (e.g., `|`). This is analogous to a boolean OR.

Parentheses can be used to group such expressions. As is usual, parentheses take precedence over the alternation operator. Table 3 shows some simple examples of operator and parenthesis usage.

[11] See `http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`

pattern	description
`fat rats | mice`	matches *fat rats* OR *mice*
`fat (rats | mice)`	matches *fat rats* OR *fat mice*

Table 3: Example of parentheses usage to change operator precedence.

Odin also supports several types of quantifiers (see Table 4 for details). The `?`, `*` and + postfix quantifiers are used to match a pattern zero or one times, zero or more times, and one or more times respectively. These are greedy quantifiers, and can be turned lazy by appending a question mark (e.g., `??`, `*?`, `+?`). Figure 2 illustrates the difference between greedy and lazy quantifiers.

`a b c d e f c`

pattern	match
`[]+ c`	a b c d e f c
`[]+? c`	a b c

Figure 2: Comparison of greedy (default behavior) and lazy (`?`) quantifiers.

Ranged repetitions can be specified by appending `{n,m}` to a pattern, which means that the pattern should repeat at least n times and at most m. If n is omitted (e.g., `{,m}`) then the pattern must repeat zero to m times. If m is omitted (e.g., `{n,}`) then the pattern must repeat at least n times. Ranges are greedy, and can be turned lazy by appending a question mark (e.g., `{n,m}?`, `{,m}?`, `{n,}?`) For an exact number of repetitions the `{n}` suffix is provided. Since this is an exact repetition there are no greedy/lazy variations.

Table 4 summarizes this set of quantifiers.

Symbol	Description	Lazy form
`?`	The quantified pattern is optional.	`??`
`*`	Repeat the quantified pattern *zero* or more times.	`*?`
`+`	Repeat the quantified pattern *one* or more times.	`+?`
`{n}`	Exact repetition. Repeat the quantified pattern n times.	
`{n,m}`	Ranged repetition. Repeat the quantified pattern between n and m times, where $n < m$.	`{n,m}?`
`{,m}`	Open start ranged repetition. Repeat the quantified pattern between 0 and m times, where $m > 0$.	`{,m}?`
`{n,}`	Open end ranged repetition. Repeat the quantified pattern at least n times, where $n > 0$.	`{n,}?`

Table 4: An overview of the quantifiers supported by Odin's token patterns.

Quantifiers apply either to a single token constraint or to a group of token constraints. Groups are specified by using parentheses. An example of a token pattern that uses quantifiers is shown on Example 8. This example also shows that one can use mention captures in the quantified groups (the `Number` argument), and that the captured mentions can share the same name. This is useful for the extraction of enumerations of unknown length.

```
The numbers 4, 8, 15, 16, 23 and 42 frequently recurred in Lost.
```

```
# First, find numbers by inspecting the POS tag.
# Note that this is not the only way to check for a number,
# there are other options, such as [word=/\d+/]
- name: numbers
  label: Number
  priority: 1
  type: token
  pattern: |
    [tag=CD]

# Second, match comma separated lists of numbers optionally followed
# by the word 'and' and a final number.
- name: list
  label: ListOfNumbers
  priority: 2
  type: token
  pattern: |
    @num:Number ("," @num:Number)+ (and @num:Number)?
```

Example 8: Example showcasing quantifiers and mention captures.

4.3.7 Zero-width Assertions

Zero-width assertions allow one to verify whether or not a pattern is present without including it in the matched result. Odin supports the following zero-width assertions:

Symbol	Description	Limitation
`^`	beginning of sentence	
`$`	end of sentence	
`(?=...)`	postive lookahead	
`(?!...)`	negative lookahead	
`(?<=...)`	positive lookbehind	The length of the lookbehind assertion matches must be known at compile time. This restricts the patterns supported by lookbehinds to token constraints and the exact range quantifier (e.g., `{n}`). Parentheses are supported.
`(?<!...)`	negative lookbehind	

Table 5: An overview of the zero-width assertions supported by Odin. These patterns do not consume tokens, but are useful to match patterns preceding/following the expression of interest.

4.3.8 Output

The output of any Odin rule is called a "mention", and they are actual instances of a `Mention` Scala class, or one of its subclasses (see Section 5).

The inclusion of named captures in a token pattern affects the type of `Mention` that is produced. In general, the result of applying a token pattern successfully is usually a `TextBoundMention` (see Section 5). However, if the token pattern includes named captures, then the result is a `RelationMention`, which is essentially a collection of named captures, or "arguments" (but without a predicate, or "trigger", which is typical of event mentions!). In other words, relation mentions are not dependent on a particular predicate. If one of the named captures has the name "trigger" (case insensitive), then Odin assumes that this pattern defines an event, and the result is an event mention (an instance of the `EventMention` class). Examples 9 and 10 show two simple patterns that produce an event mention and a relation mention, respectively.

Oscar lives in a **trash can.**

```
- name: event_mention_out
  label: LivesIn
  priority: 2
  type: token
  pattern: |
    (?<resident>Oscar)
    (?<trigger>[lemma=live])
    in [tag=DT]? (?<location>[tag=/^N/]+)
```

Example 9: An example of a token pattern rule that produces an event mention through the specification of a trigger.

Dr. Frankenstein spends a lot of time in the graveyard.

```
- name: relation_mention_out
  label: PersonWithTitle
  priority: 2
  type: token
  pattern: |
    (?<title>[word=/(?i)^mr?s|dr|prof/]) @person:Person
```

Example 10: An example of a token pattern rule that produces a relation mention. This rule has named arguments, but does not specify a trigger. For brevity, we assume that `Person` mentions have already been identified.

4.4 Dependency Patterns

While token patterns are quite powerful, they are, of course, not too robust to syntactic variation. Writing patterns over syntactic structure produces generalizations with broader coverage that do not sacrifice precision. Consider the sentences in Figure 3:

Noam danced at midnight with the **leprechaun.**

Noam, in full view of the three-legged robot, **danced** at dawn with the **leprechaun.**

Noam danced under the moonlight at midnight with the **leprechaun.**

His friends watched in awe while **Noam danced** the forbidden jig with the **leprechaun** at midnight.

Figure 3: These sentences show some of the infinite syntactic variation describing a dance between two entities.

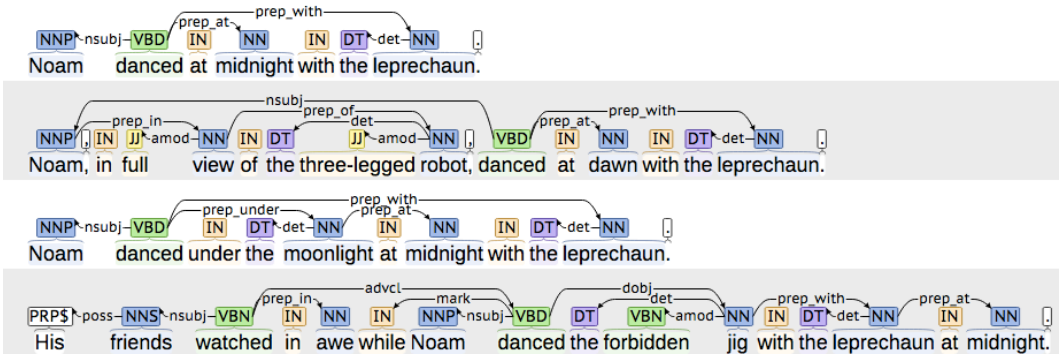

Figure 4: The relational-dependency parse for the sentences in Figure 3.

While it requires several token-pattern rules to precisely capture the syntactic variation shown in Figure 3, all of these variants can be covered with a single rule using a dependency pattern (see Example 11).

```
- name: dancers_1
  label: Dance
  priority: 2
  pattern: |
    trigger = [lemma=dance]
    dancer:Entity = nsubj
    partner:Entity = dobj? prep_with
```

Example 11: A dependency rule that expects two arguments: (1) a nominal subject and (2) the head word complements of a "with" prepositional phrase off of the lemmatized trigger, *dance*; (2) may be preceded by an optional hop through a direct object (`dobj`) relation. Note the optional hop through a direct object (`dobj`). Parsers often struggle with prepositional attachment, so we have added an optional `dobj` in this rule to be robust to such errors.

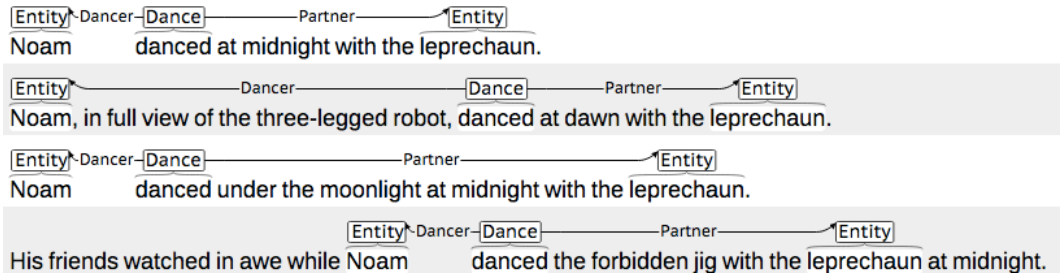

Figure 5: The structured output of the rule in Example 11.

Formally, a dependency pattern describes a traversal over a syntactic dependency graph. Again, we currently use Stanford dependencies de Marneffe and Manning [2008] in Odin, but Odin is independent of the representation used. Odin's dependency patterns are composed of several fields. To boot, dependency patterns defining event rules require a "trigger" that must be set to a token pattern (see previous section). This token pattern describes a valid predicate for the event of interest. The rest of the fields are event arguments defined through a syntactic path from the trigger to some mention (entity or event) that was previously matched by another rule. The path is composed of *hops* and optional *filters*. The hops are edges in the syntactic dependency graph; the filters are token constraints on the nodes (tokens) in the graph.

Hops can be *incoming* or *outgoing*. An *outgoing* hop follows the direction of the edge from HEAD →DEPENDENT; an *incoming* hop goes against the direction of the edge, leading from DEPENDENT →HEAD. For example, the dependency "jumped" → "Fonzie" is outgoing ("jumped" is the head), but it is considered incoming when traversed in the other direction: "Fonzie" ← "jumped".

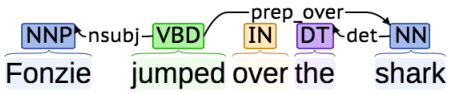

Figure 6: A simple sentence with its corresponding dependency parse tree.

```
  pattern: |
    trigger = [lemma=jump]
    entity:Noun = nsubj
    obstacle:Noun = prep_over
```

Example 12: A simple, two-argument dependency pattern composed solely of outgoing hops, which matches the "jumping" event above. We are assuming that a different rule created a `Noun` mention for every `NN*`.

An outgoing dependency is matched using the > operator followed by a string matcher, which operates on the label of the corresponding dependency, e.g., `>nsubj`. Because most patterns use outgoing hops, (i.e. HEAD →DEPENDENT), the > operator is implicit and can therefore be omitted. An incoming relation (i.e. DEPENDENT →HEAD) is matched using a required < operator followed by a string matcher. >> is a wildcard operator that can be used to match any outgoing dependency. << is a wildcard operator that can be used to match any incoming dependency.

Importantly, restrictions may be imposed on the nodes (i.e., tokens) visited when following dependencies, using the usual token constraints. Example 13 illustrates such constraints on both the `robber` (using the POS tag) and the `property` (using the actual word).

Gonzo stole her **chicken.**

Gonzo stole her ~~heart~~**.**

```
- name: np
  label: Noun
  priority: 1
  type: token
  unit: tag
  pattern: |
    /^N/

- name: steal-1
  label: Steal
  priority: 2
  pattern: |
    trigger = [lemma=steal]
    robber:Noun = nsubj [tag=NNP] # We are only interested in Proper Nouns
    property:Noun = dobj [!word=heart] # Let's keep the romance out of it.
```

Example 13: While these two sentences are syntactically identical, only one pertains to theft of tangible goods. We are assuming that a different rule created a `Noun` mention for every `NN*`.

Just as in token patterns, dependency patterns can include parentheses and the alternation operator `|`. For example, the pattern `nsubj|agent` matches an outgoing dependency whose label is either `nsubj` or `agent`.

4.4.1 Named Arguments for Dependency Patterns

Clearly, naming event arguments is important (e.g., one may want to keep track who is the agent and who is the patient in a robbery event). We probably already observed that Odin has a simple syntax for this: a path to an argument begins with `name:label = path`, where `label` is the the label of an existing `Mention`. The path must lead to a token that is a part of a `Mention` with the specified label. Argument names are required and *unique*, i.e., you can't have two different patterns with the same name. But the same pattern may match multiple mentions! If, for example, an argument with the name "theme" matched three different entities, then three event mentions will be generated, each with one entity as the *theme*. If the given path to the *theme* fails to match any `Mention`, then no event mentions will be created.

At times one may need to make an argument optional or allow for more than one argument with the same name in a single event mention. This can be achieved through the use of argument quantifiers. Arguments can be made optional with the `?` operator. The + operator is used to indicate that a single event mention with all matches should be created. The $\star$ is similar to + but also makes the argument optional. If the exact number of arguments with the same name is known, it can be specified using the exact repetition quantifier $\{k\}$. In cases of exact repetitions, the cartesian product will be applied to the `Mentions` matching the given path. If k `Mentions` are asked for in a path p and n are found to match p, then j event mentions will be produced, where j is the binomial coefficient shown in Equation 4.4.1. A few rules using these argument quantifiers are shown in Examples 14, 15, & 16.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{1}$$

Cities like **London, Paris, Tokyo,** and **Beijing.**

Figure 7: The sentence used by the rules shown in Examples 14, 15, & 16. We are assuming that a different rule created a `Location` mention for every location NE.

```
- name: cities-1
  label: Cities
  priority: 2
  pattern: |
    trigger = [lemma=city]
    # produces 4 EventMentions each with 1 city
    city:Location = prep_like
```

Example 14: An example of a complete dependency pattern rule without an argument quantifier.

Mention	Cities
1	London
2	Paris
3	Tokyo
4	Beijing

Table 6: The four mentions produced by the dependency pattern shown in Example 14.

```
    trigger = [lemma=city]
    # produces 1 EventMention with 4 cities
    city:Location+ = prep_like
```

Example 15: An example of a dependency pattern with a + argument quantifier. Its output is shown in Table 7.

Mention	Cities
1	London, Paris, Tokyo, Beijing

Table 7: The single mention produced by the rule shown in Example 15.

```
1    trigger = [lemma=city]
2    # produces 6 EventMentions each with 2 cities
3    city:Location{2} = prep_like
```

Example 16: An example of a dependency pattern with a {k} quantifiers on event arguments. The scattering effect of the $\{k\}$ quantifier is shown in Table 8.

Mention	Cities
1	London, Paris
2	London, Tokyo
3	London, Beijing
4	Paris, Tokyo
5	Paris, Beijing
6	Tokyo, Beijing

Table 8: The six mentions produced by the rule shown in Example 16.

4.4.2 Quantifiers for Dependency Patterns

In addition of the above quantifiers on event arguments, Odin supports quantifiers inside the actual dependency patterns. They are shown in Table 9.

The `?`, `*` and + postfix quantifiers are used to match a pattern zero or one times, zero or more times, and one or more times respectively. There is no notion of greedy/lazy dependency patterns.

Ranged repetitions can be specified by appending `{m,n}` to a pattern, and means that the pattern should repeat at least m times and at most n. If m is omitted (e.g., `{,n}`) then the pattern must repeat zero to n times. If n is omitted (e.g., `{m,}`) then the pattern must repeat at least m times. There is no notion of greedy/lazy dependency patterns. For an exact number of repetitions the `{n}` suffix is provided.

For example, the pattern `/prep/+` matches a sequence of 1 or more outgoing dependencies whose labels contain `prep`. The pattern `dobj* /prep/{,3}` matches 0 or more `dobj` dependencies, followed by up to 3 outgoing dependencies that contain `prep`.

Symbol	Description
`?`	The quantified pattern is optional.
`*`	Repeat the quantified pattern *zero* or more times.
+	Repeat the quantified pattern *one* or more times.
`{n}`	Exact repetition. Repeat the quantified pattern n times.
`{n,m}`	Ranged repetition. Repeat the quantified pattern between n and m times, where $n < m$.
`{,m}`	Open start ranged repetition. Repeat the quantified pattern between 0 and m times, where $m > 0$.
`{n,}`	Open end ranged repetition. Repeat the quantified pattern at least n times, where $n > 0$.

Table 9: An overview of the quantifiers supported by Odin's dependency patterns.

4.4.3 Zero-width Assertions

For dependency patterns, there no lookbehind or lookahead assertions, only lookaround assertions. The lookaround syntax is `(?= pattern)` for positive assertions and `(?! pattern)` for negative assertions. Example 17 shows an example of a positive lookaround in action.

Dennis crashed his mom's **car.**

Dennis crashed his ~~car~~.

```
- name: np
  label: Noun
  priority: 1
  type: token
  unit: tag
  pattern: |
    /^N/

- name: accident-1
  label: Accident
  priority: 2
  pattern: |
    trigger = [lemma=crash]
    agent:Noun = nsubj [tag=NNP] # We are only interested in Proper Nouns
    # Only match if this is mom's car
    vehicle:Noun = dobj (?= poss [lemma=mom]) [lemma=car]
```

Example 17: Sometimes ownership matters. Perhaps we want to know whether or not Dennis should be grounded for crashing a car. Did Dennis crash his mother's car? A positive lookaround is needed for this.

4.4.4 Output

The result of applying a dependency pattern successfully is usually an event mention. If a trigger is not specified, a relation mention is produced (see Figures 18 & 19 for details).

Oscar lives in a **trash can.**

```
- name: dep_event_mention_out
  label: LivesIn
  priority: 2
  pattern: |
    trigger = [lemma=live]
    resident:Person = nsubj
    location:Location prep_in
```

Example 18: An example of a dependency pattern rule that produces an event mention through the specification of a trigger.

Dr. Frankenstein spends a lot of time in the graveyard.

```
- name: sometitle-1
  label: Title
  priority: 1
  type: token
  pattern: |
    [word=/(?i)^mr?s|dr|prof/]

- name: dep_relation_mention_out
  label: PersonWithTitle
  priority: 2
  pattern: |
    person:Person
    title:Title = nn
```

Example 19: An example of a dependency pattern rule that produces a relation mention. This rule has named arguments, but does not specify a trigger. When the trigger field is omitted in a dependency pattern, the first field given should specify a named argument using the mention retrieval syntax (`argname:MentionLabel`). All subsequent dependency patterns used by the other arguments are anchored on this first argument.

4.5 Building a Grammar

By now, we hope you are somewhat confident that you can write Odin rules. Of course, the next step is to put them together into a complete grammar. This can be very simple: minimally, all you have to do is to store them all into a single file which is then loaded into an Odin engine (see Section 7). If you care a lot about efficiency, you can tune your grammar by assigning priorities to rules. For example, rules that match entities should be executed before (i.e., have a lower priority) than rules that match events where these entities serve as arguments. (But again, this is not needed: Odin takes care of pipelining rules internally.)

But some domain grammars are more complicated than a simple sequence of rules. You may have event labels that are so complex that you would prefer to store them in a taxonomy. Some event types have almost exactly the same syntactic representations as others, so you would like to reuse some rules. Odin supports all these issues. We describe them next.

4.5.1 Master File

The master file is a grammar's entry point, or the file is passed to the Odin runtime engine. As mentioned, for simple grammars, this file can be simply a collection of rules. For more complicated scenarios, this file must contain a required `rules` section, and two optional sections: `taxonomy` and `vars`. Let us describe these sections next.

4.5.2 Taxonomy

The taxonomy is a forest (meaning a collection of trees) of labels that, if specified, is used by Odin as the hierarchy for mention labels. An example taxonomy is shown in Example 20.

```yaml
# a tree hierarchy can be used to define the taxonomy
- organism:
  - prokaryotic:
    - archaebacteria
    - eubacteria
  - eukaryotic:
    - unicellular:
      - protista
    - multicellular:
      - autotrophic:
        - plantae
      - heterotrophic:
        - fungi
        - animalia
# we want to include robots in our taxonomy
# but they are not organisms, what can we do?
# fortunately, multiple trees are supported
- robot
```

Example 20: Example taxonomy

If a taxonomy is provided, then all the labels used by the rules must be declared in the taxonomy. This is obviously useful for catching typos. More importantly, the taxonomy hierarchy is used to robustly match mentions in subsequent rules. For example, if a rule creates an entity mention with the label `animalia` from the taxonomy in Example 20, this mention will be matched as argument in a subsequent rule, which requires that argument to be of label `multicellular`. This is because `animalia` is a hyponym of `multicellular`, i.e., it is directly derived from it.

If the value of `taxonomy` is a single string, then it will be interpreted as a file name and the taxonomy will be read from that file. It should be noted that the taxonomy may only be specified in the master file, whether included directly or provided through an import (see Example 21).

```yaml
# the taxonomy file should contain only the contents of the taxonomy (without the
    taxonomy: section name)
taxonomy: path/to/taxonomy.yml
```

Example 21: An example of a taxonomy import.

4.5.3 Variables and Templates

It is very common that similar events share the same syntactic structure. For example, in the biomedical domain, all the biochemical reactions (there are between 10 and 20 of these) share the same structure. For example, "A phosphorylates B" is similar to "A ubiquitinates B", with the exception of the predicate: "phosphorylates" vs. "ubiquitinates". In such situations, we would like to reuse these syntactic structures between events (so we do not write the same rules 10–20 times). Odin supports these through the use of variables and rule templates, where rule templates are simply rules that use variables. For example, one could write a single rule template for the above example, where the trigger constraints are encoded using a variable.

In general, variables can be declared as a YAML mapping, and they can be substituted in rules using the `${variableName}` notation (see Examples 22 & 23). Furthermore, wherever a rule can be specified, you can also import a file, through the `import` command, and its optional `vars` parameter. This gives one a further opportunity to instantiate variables. Example 22 shows the `import` command in action.

```yaml
# global variables
vars:
  myTrigger: "eat"

rules:
  # import rules from file
  # if variables are used in the imported file,
  # they will be retrieved from the global vars
  - import: path/to/template.yml

  # import rules from file
  # myTrigger is overridden for this import
  - import: path/to/template.yml
    vars:
      myTrigger: "sell"

  # rules and imports can live together in harmony :)
  - name: somedude
    label: Person
    type: token
    pattern: |
      [entity='B-Person'] [entity='I-Person']*
```

Example 22: An example of a master file that uses import statements and demonstrates variable precedence. Note that variables can be instantiated in three different places: (a) in the template file itself, (b) when the `import` command is used, or (c) at the top of the master file. The precedence is: (b) > (c) > (a). For this particular example, it means that the value chosen for the `myTrigger` variable is "eat" for the first import (LINE 9) and "sell" for the second import (LINE 13).

```yaml
vars:
  # these variables are superseded by those in the master file
  myTrigger: "buy"
  myMentionLabel: "Food"

rules:
 - name: example_rule
   label: Event
   type: token
   priority: 1
   pattern: |
     @person:Person # match a person
     (?<trigger> [lemma=${myTrigger}]) # trigger comes from provided variable
     [tag="DT"]? @object:${myMentionLabel} # retrieve mention with given label
```

Example 23: The `template.yml` file imported in Example 22.

5 Mentions, or the Output of Rules

As hinted before in this document, each rule produces a `Mention` object when it successfully matches some text. These objects are nothing magical: we simply use them to store everything that the rule contains, and the corresponding text matched. Table 10 summarizes the fields of the mention object.

Field	Description
`labels`	The sequence of labels to associate with a mention.
`tokenInterval`	The open interval token span from the first word to the final word $+1$.
`startOffset`	The character index in the original text at which the mention begins.
`endOffset`	The character index in the original text at which the mention ends $+1$.
`sentence`	The sentence index of this mention.
`document`	The document (composed of annotated sentences) from which this mention originates.
`arguments`	A map from argument name (a `String`) to a sequence of mentions.
`foundBy`	The name of the rule that "found" this mention.

Table 10: An overview of the most important fields of the `Mention` class.

Sometimes, actual code is best at explaining things. We highly encourage the reader to take a look at the code implementing `Mention` and its subclasses[12]. Note that some the information stored in mentions, e.g., the token interval of the mention, refer to data structures produced by our preprocessing code, such as `Sentence` and `Document`. Again, reading through the code is the best way to learn about these[13].

5.1 TextBoundMention

The `Mention` class is subclassed by several other classes. The simplest is `TextBoundMention`. A `TextBoundMention` is created when the output of a rule is a flat structure, i.e., a contiguous sequence of tokens in a sentence. More formally, a `TextBoundMention` will have a `tokenInterval`, but its `arguments` map will be empty. These mentions are usually used to represent entities or event triggers.

5.2 RelationMention

A `RelationMention` encodes n-ary relations between its arguments. All the arguments are named (based on the argument names specified in the matched rule), and are stored in the `arguments` map. Importantly, several arguments may have the same name! This is extremely useful when one needs to capture in a rule enumerations of several valid arguments with the same role (for example, a `food` argument may capture multiple foods consumed at a dinner).

5.3 EventMention

An `EventMention` is similar to a `RelationMention`, with just one additional feature: it has a `TextBoundMention` that represents the trigger of the event. In other words, the `arguments` map contains an additional argument, labeled `trigger`, which stores the event's predicate. Note: an event must have exactly one trigger.

[12] `https://github.com/clulab/processors/blob/master/src/main/scala/edu/arizona/sista/odin/Mention.scala`

[13] `https://github.com/clulab/processors/blob/master/src/main/scala/edu/arizona/sista/processors/Document.scala`

6 Advanced: Customizing Rule Output with Actions

Note: you are welcome to skip this section. We expect only a small numbers of users, who need deep customization of Odin, to find this section necessary.

As described in the previous section, Odin rules produce mentions, which store all the relevant information generated during the match. This is sufficient for most common usages of Odin, but sometimes this information requires some changes. For example, one could use coreference resolution to replace event arguments that are pronouns with their nominal antecedents, as indicated by the coreference resolution component. This is not easily done though rules, and this is when actions become necessary.

Actions are Scala methods (implemented by the domain developer!) that can be applied by Odin's runtime engine to the resulting `Mention`s after matching the rule. An `Action` has the type signature shown in Example 24.

```
1 def action(mentions: Seq[Mention], state: State): Seq[Mention]
```

Example 24: Signature of action methods.

A rule will first try to apply its pattern to a sentence. Any matches will be sent to the corresponding action as a `Mention` sequence. If an action is not explicitly named, the default identity action will be used, which returns its input mentions unmodified (i.e. the input's identity).Actions receive as input parameters this `Mention` sequence and also the runtime engine's `State`.

The `State` object (second parameter) provides read-only access to *all* the mentions previously created by Odin in the current document. This information may be useful to implement global decisions, e.g., coreference resolution across the entire document.

Actions must return a `Mention` sequence that will be added to the `State` at the end of the current iteration by the runtime engine. For example, the simplest possible action would return the `mentions` it received as the first input parameter. Example 25 shows an only slightly more complicated action that removes any `Mention` containing the text "Fox".

```
1 def action(mentions: Seq[Mention], state: State): Seq[Mention] = {
2     mentions.filter(_.text contains "Fox")
3 }
```

Example 25: An example of an `action` that removes any `Mention` containing the text "Fox".

Note that, in addition to attaching actions to individual rules, actions can also be called globally at the end of each iteration by the runtime engine. This means that the extractor engine (see Example 26) must receive this global action as a parameter during its construction.

```
1  // The simplest instantiation where no actions are specified.
2  // Here the matches produced by a our rules are returned unmodified.
3  val eeNoActions = ExtractorEngine(rules)
4  // myActions is an object containing the implementation of any actions
5  // named in the rules
6  val eeWithActions = ExtractorEngine(rules, myActions)
7  // Here we specify both an actions object and a global action
8  val eeWithActionsAndGlobal = ExtractorEngine(rules, myActions, myGlobalAction)
9  // We can also choose to specify only a global action
10 val eeWithGlobalOnly = ExtractorEngine(rules, globalAction = myGlobalAction)
```

Example 26: The `ExtractorEngine` may be instantiated in several ways.

Global actions have the same signature, but, in this context, the `mentions` parameter contains all mentions found in this iteration of the engine. Any mentions produced by rule-local actions will only make it into the `State` iff they pass successfully through the global actions. By default, the global action returns its input unmodified (i.e. the input's identity).

7 Putting it Together: the Odin API

In the previous sections we learned how to write token and dependency patterns using the Odin information extraction framework. In this section, we will go through the set up of a complete system using Odin to extract marriage events from free text. In Example 27, we define a minimal grammar which we assume to be saved to the current working directory in a file named `marriage.yml`.

```
- name: ner-person-or-pronouns
  label: Person
  priority: 1
  type: token
  pattern: |
    # This pattern uses the output of an NER system to
    # create a Person mention
    [entity=PERSON]+
      |
    # We will also consider some pronouns to be person Mentions
    [lemma=/^he|she|they/]

- name: ner-date
  label: [Date]
  priority: 1
  type: token
  pattern: |
    [entity=DATE]+

- name: ner-loc
  label: Location
  priority: 1
  type: token
  pattern: |
    [entity=LOCATION]+

# optional location and date
- name: marry-syntax-1
  label: Marriage
  priority: 2
  example: "He married Jane last June in Hawaii."
  type: dependency
  pattern: |
    # avoid negative examples by checking for "neg" relation
    trigger = [lemma=marry & !outgoing=neg]
    spouse:Person+ = (<xcomp? /^nsubj/ | dobj) conj_and?
    date:Date? = /prep_(on|in|at)/+ | tmod
    location:Location* = prep_on? /prep_(in|at)/+
```

Example 27: An example of a small set of rules designed to capture a marriage event and its participants. The rules that run in priority 1 make use of the output of an NER system to capture mentions for `Person`, `Location`, and `Date`. According to the rule "marriage-syntax-1", a `Marriage` event requires at least one spouse and may optionally have a `Date` and `Location`.

We can now use our `marriage.yml` event grammar to extract mentions from free text. Example 28 shows a simple Scala program to do just this. We instantiate a `CoreNLPProcessor` which uses Stanford's CoreNLP to parse and annotate the provided text with the attributes required by Odin (see Table 2 for a list of the relevant attributes). This annotated text is stored in a `Document`

which is then passed to Odin through the `EventEngine.extractFrom()` method. Finally we collect the `Marriage` mentions found by Odin and display them using the `Mention.json()` method, which converts the mention into a `JSON` representation. A portion of this output is shown in Example 29.

```
import edu.arizona.sista.odin._
import edu.arizona.sista.processors.corenlp.CoreNLPProcessor

object SimpleExample extends App {
  // two example sentences
  val text = """|John and Alice got married in Vegas last March.
                |Caesar and Cleopatra never married.
                |I think they got married.
                |Zarbon and Frederick will marry next summer.
                |She and Burt finally got married.
                |Simon and Samantha got married in Tucson on March 12, 2010 at the Desert Museum.
                |""".stripMargin

  // read rules from general-rules.yml file in resources
  val source = io.Source.fromFile("marriage.yml"))
  val rules = source.mkString
  source.close()

  // Create a simple engine without custom actions
  val extractor = ExtractorEngine(rules)

  // annotate the sentences
  val proc = new CoreNLPProcessor
  val doc = proc.annotate(text)

  // extract mentions from annotated document
  val mentions = extractor
                 .extractFrom(doc)
                 // only keep the Marriage mentions
                 .filter(_ matches "Marriage")

  // display the mentions
  mentions.foreach{ m => println(m.json(pretty=true)) }

}
```

Example 28: A simple Scala program using the `marriage.yml` rules shown in Example 27. These rules do not call any custom actions. For an explanation of how to link rules to custom actions, please refer to Section 6.


```
{
  "type":"Event",
  "labels":["Marriage"],
  "sentence":5,
  "foundBy":"marry-syntax-1",
  "trigger":{
    "type":"TextBound",
    "tokenInterval":[4,5],
    "characterOffsets":[212,219],
    "labels":["Marriage"],
    "sentence":5,
    "foundBy":"marry-syntax-1"
  },
  "arguments":{
    "spouse":[{
      "type":"TextBound",
      "tokenInterval":[0,1],
      "characterOffsets":[189,194],
      "labels":["Person"],
```

```
      "sentence":5,
      "foundBy":"ner-person-or-pronouns"
    },{
      "type":"TextBound",
      "tokenInterval":[2,3],
      "characterOffsets":[199,207],
      "labels":["Person"],
      "sentence":5,
      "foundBy":"ner-person-or-pronouns"
    }],
    "date":[{
      "type":"TextBound",
      "tokenInterval":[8,12],
      "characterOffsets":[233,247],
      "labels":["Date"],
      "sentence":5,
      "foundBy":"ner-date"
    }],
    "location":[{
      "type":"TextBound",
      "tokenInterval":[6,7],
      "characterOffsets":[223,229],
      "labels":["Location"],
      "sentence":5,
      "foundBy":"ner-loc"
    },{
      "type":"TextBound",
      "tokenInterval":[14,16],
      "characterOffsets":[255,268],
      "labels":["Location"],
      "sentence":5,
      "foundBy":"ner-loc"
    }]
  }
}
```

Example 29: An example of one of the captured `Marriage` mentions outputted as `JSON`. The "characterOffsets" field corresponds to the original text provided in Example 28).

An example of a complete project including details on how to specify Odin's dependencies is available here:

`https://github.com/clulab/odin-examples`

Readers seeking a starting point for their own projects can refer to the code in the linked repository which contains working examples covering both simple and complex scenarios.

References

Douglas E Appelt and Boyan Onyshkevych. The common pattern specification language. In *Proc. of the TIPSTER Workshop*, pages 23–30, 1998.

Douglas E. Appelt, Jerry R. Hobbs, John Bear, David Israel, and Mabry Tyson. Fastus: A finite-state processor for information extraction from real-world text. In *Proceedings of the International Conferences on Artificial Intelligence (IJCAI)*, 1993.

Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml) version 1.1. *yaml. org, Tech. Rep*, 2005.

Angel X. Chang and Christopher D. Manning. TokensRegex: Defining cascaded regular expressions over tokens. Technical Report CSTR 2014-02, Computer Science, Stanford, 2014.

Laura Chiticariu, Yunyao Li, and Frederick R Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proc. of EMNLP*, 2013.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Developing Language Processing Components with GATE (Version 6)*. University of Sheffield, 2011.

Marie-Catherine de Marneffe and Christopher D. Manning. The Stanford typed dependencies representation. In *Proc. of COLING Workshop on Cross-framework and Cross-domain Parser Evaluation*, 2008.

Lawrence Hunter, Zhiyong Lu, James Firby, William A Baumgartner, Helen L Johnson, Philip V Ogren, and K Bretonnel Cohen. Opendmap: an open source, ontology-driven concept analysis engine, with applications to capturing knowledge regarding protein transport, protein interactions and cell-type-specific gene expression. *BMC bioinformatics*, 9(1):78, 2008.

Roger Levy and Galen Andrew. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *Proc. of LREC*, 2006.

C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proc. of ACL*, 2014.

Lance A Ramshaw and Mitchell P Marcus. Text chunking using transformation-based learning. *arXiv preprint cmp-lg/9505040*, 1995.