Odin's Runes: A Rule Language for Information Extraction

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, Mihai Surdeanu
University of Arizona
Tucson, AZ 85721, USA
{marcov, hahnpowell, msurdeanu}@email.arizona.edu

Abstract

Odin is an information extraction framework that applies cascades of finite state automata over both surface text and syntactic dependency graphs. Support for syntactic patterns allow us to concisely define relations that are otherwise difficult to express in languages such as Common Pattern Specification Language (CPSL), which are currently limited to shallow linguistic features. The interaction of lexical and syntactic automata provides robustness and flexibility when writing extraction rules. This paper describes Odin's declarative language for writing these cascaded automata.

Keywords: rule-based, information extraction, cascade of finite state automata

1. Introduction

We recently released Odin (Open Domain INformer), a novel rule-based information extraction (IE) framework (Valenzuela-Escárcega et al., 2015b). At the core of this framework is Odin's Runes, our rule grammar language. The core feature of this language is supporting different types of rules, e.g., operating over surface or syntactic structures, which can interact in the same grammar.

At a high level, the design of this rule language follows the simplicity principles promoted by other natural language processing (NLP) 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). In particular, we aimed for the following desirable characteristics:

Simplicity: The language extends familiar concepts from regular expressions and context free grammars.

Expressivity: The rules capture complex constructs when necessary, such as: (a) nested structures, and (b) complex regular expressions over syntactic patterns for event arguments.

Robustness: To recover from unavoidable syntactic errors, syntactic patterns can be used alongside token-based surface patterns that incorporate shallow linguistic features.

Extensibility: The language is designed to be modular, i.e., new types of rules can be easily added to the language. We currently support rules based on syntactic and surface structures, and we plan extensions over abstract meaning representation (AMR) (Banarescu et al., 2012) and semantic roles (Surdeanu et al., 2008). Importantly, all of these types of rules can operate within the same grammar.

In this paper we summarize Odin's Runes. However, given space limitations, this description is likely to be incomplete. We recommend that interested readers examine the full manual (Valenzuela-Escárcega et al., 2015a), which has been made available as an arXiv document[1].

[1] http://arxiv.org/abs/1509.07513v1

2. Related Work

Since the advent of FASTUS (Appelt et al., 1993), most rule-based IE frameworks implement architectures relying on a cascade of finite state automata (FSA). This approach has proven capable of producing fast and robust parsers for unstructured text (Abney, 1996). The success of FSA cascades continues even today with systems such as GATE (Cunningham et al., 2002).

FASTUS introduced the Common Pattern Specification Language (CPSL) as a formalism for specifying cascaded FSA grammars (Appelt and Onyshkevych, 1998). A grammar in CPSL is specified by defining a cascade of finite state transducers that work by matching regular expressions over the lexical features of the input symbols.

Other languages that follow CPSL's approach of matching regular expressions over the lexical features of the input are GATE's Java Annotation Patterns Engine (JAPE) (Thakker et al., 2009), Stanford's TokensRegex (Chang and Manning, 2014), and the Allen Institute for Artificial Intelligence taggers[2]. Odin follows in this lineage; however, unlike these approaches, Odin allows the mixing of both surface- and syntax-based rules in the same grammar. Furthermore, because Odin builds on top of simple and proven syntactic dependency representations (De Marneffe and Manning, 2008a), the learning curve for Odin's Runes is short.

SProUT's XTDL (Piskorski et al., 2004) extends CPSL's approach using unification-based grammars to give the language more expressivity. However, this introduces additional complexity in the language. In our opinion, this is not always necessary in domain-specific scenarios, where lexical information fully disambiguates the context. Furthermore, similar to most previous work, XTDL does not support syntactic patterns.

From the languages that support syntax, Stanford's Tregex matches patterns over constituency trees (Levy and Andrew, 2006). For Odin's Runes we chose to use dependency-based syntax for two reasons: simplicity of representation, and availability of linear-time parsers (Chen

[2] https://github.com/allenai/taggers

WTX inhibits the ubiquitination of **NRF2**.

Figure 1: A sentence containing two events in the biomedical domain: a ubiquitination, and a negative regulation. Bold text denotes biochemical entities previously identified by an NER system.

```
- name: ner
  label: Protein
  type: token
  pattern: |
    # Named Entity labels in the IOB style
    [entity="B-Protein"][entity="I-Protein"]*

- name: ubiq-surf
  label: [Ubiquitination, Event]
  type: token
  pattern: |
    # a single-token trigger
    (?<trigger>ubiquitination)
    # the theme must be a Protein
    of @theme:Protein
    # the cause of might not be specified
    (by @cause:Protein)?

- name: negreg-surf
  label: [Negative_regulation, Event]
  type: token
  pattern: |
    @cause:Protein
    # any conjugation of the lemma "inhibit"
    (?<trigger>[lemma=inhibit & tag=/^V/])
    # an optional determiner
    [tag=DT]?
    # the theme of this event is another Event
    @theme:Event
```

Example 1: Rules that capture the events shown in Figure 1. All the rules use surface patterns.

and Manning, 2014). Semgrex is a language that modifies Tregex to operate over dependency graphs (Chambers et al., 2007)[3]. However, neither of these languages support cascaded FSA.

In a departure from CPSL, IBM's SystemT is a rule-based IE system that uses the AQL language, which is inspired from SQL (Li et al., 2011). AQL is a powerful language that implements an IE algebra (Reiss et al., 2008). However, in our opinion, this loses some of the simplicity that Odin's Runes enjoys.

3. Walkthrough Example

In this section we show two Odin grammars in the biomedical domain as a gentle introduction to the language. Both grammars match over the sentence shown in Figure 1. All Odin grammars are encoded using YAML, which is a human-readable data serialization language (Ben-Kiki et al., 2005). YAML's readability and support for comments were the main motivations for choosing it as the format for Odin's Runes.

Example 1 lists a grammar that consists of surface patterns only. Example 2 shows a grammar that captures the same events in the example sentence, but it is implemented with syntactic rules.

The grammar in Example 1 work as follows:

- The `ner` rule converts the IOB output of an external NER tool into Odin entity mentions labeled `Protein`. In general, Odin mentions are data structures that store the output of a matched rule. For example, in this instance, the mention created by this rule captures the fact that the span of tokens from 1 to 2 (exclusive) and from 6 to 7 correspond to a named entity labeled `Protein`. In most situations, mentions are transparently created and managed by the Odin runtime system.

- The `ubiq-surf` rule matches a ubiquitination event using a surface pattern. First, the token "ubiquitination" is captured as the event trigger, followed by the token "of" and a protein mention, which is captured as the event theme. Optionally, it can be followed by the token "by" and a protein mention that would be captured as the event cause. Note that this rule defines two labels for the resulting mention, which are used to define an implicit taxonomy. For example, here `Ubiquitination` is a kind of `Event`. Taxonomies can also be explicitly defined, in which case the mention label specifies the most specific node in the taxonomy where this mention is mapped. For brevity, we omit examples with explicit taxonomies in this section. See Section 4.4. for a discussion on explicit taxonomies.

- The `negreg-surf` rule matches a negative regulation event using a similar pattern. First, a protein mention is captured as the event theme, followed by a token with the following attributes:

  1. the lemma is "inhibit'
  2. the POS tag starts with "V" (a verb).

  This token is captured as the event trigger. Then an optional determiner is matched followed by an existing event mention, which is captured as the current mention's *theme*.

At runtime, these three rules are automatically organized in a cascade, where the first rule finds the `Protein` mentions, which are then used to populate the event mention extracted by the second rule. Lastly, the third rule is executed, which uses the outputs of the first and second rules to generate a nested event.

Unlike the grammar in Example 1 which relies solely on surface patterns, the grammar in Example 2 uses mostly syntax:

- The `ner` rule is identical to the one in Example 1, and is necessary to capture the IOB output of the NER.

- The `ubiq-syn` rule matches a ubiquitination event, which is anchored around a nominal predicate (`trigger`), "ubiquitination", and has two arguments: a mandatory `theme`, which is syntactically attached to the verbal trigger through the preposition "of", and an optional `cause`, attached to the trigger through the preposition "by". The resulting event mention is assigned the `Ubiquitination` and `Event` labels.

[3]See also Semgrex's online documentation: `http://nlp.stanford.edu/software/tregex.shtml`

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

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

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

Example 2: Rules that capture the events shown in Figure 1. The first rule uses a surface pattern, while the other two use syntactic patterns.

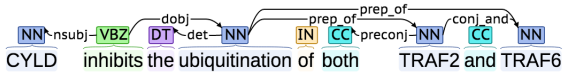

Figure 2: A sentence that can't be completely handled by the grammar in Example 1 but can be handled by the grammar in Example 2.

- The `negreg-syn` rule implements a negative regulation driven by a verbal predicate. Note that one of the arguments is an event produced by the `ubiq-syn` rule. As discussed, the Odin runtime guarantees that the latter rule completes before the former.[4]

Although both example grammars capture the same output for the sentence shown in Figure 1, the syntax-based one is considerably more general. For example, the syntax-based grammar correctly finds two ubiquitination events and two negative regulations in the sentence "CYLD inhibits the ubiquitination of both TRAF2 and TRAF6" because the dependency graph correctly connects "ubiquitination" to "TRAF2" and "TRAF6", as seen in Figure 2. On the other hand, the surface-based grammar misses the ubiquitination event involving "TRAF6" (and the negative regulation of this ubiquitination), because the last two tokens of the sentence are not explicitly handled by the rules. However, syntax-based grammars assume that a syntactic parser is available and produces robust output. This is not always true, especially in domain-specific settings. In such situations it is beneficial to mix syntax and surface rules, or rely solely on the latter.

4. Details of the Rule Language

The previous example shows some of Odin's capabilities, but Odin's Runes is considerably more powerful. In this section, we detail what we consider to be the most relevant language features. We begin with a description of the main features of Odin's dependency patterns[5].

4.1. Syntactic Dependency Patterns

To mitigate language sparsity, Odin provides the capability to match patterns over a sentence's dependency graph (de Marneffe and Manning, 2008b). With these patterns, Odin captures event or relation structures. Event structures are composed of a predicate and its corresponding arguments, and relation structures are only composed of arguments (no predicate).

When we want to retrieve an event, the predicate, or *trigger*, is defined using a surface pattern over sequences of tokens and their associated attributes, such as a word's lemma form or its part-of-speech (POS) tag. Event arguments are identified by dependency paths anchored at the matched trigger. These arguments have semantic constraints represented as labels (e.g., `Protein` or `Event` in the `negreg` rule in Example 2).

Syntactic patterns for relations between mentions are also supported by first specifying a previously found mention as an anchor; the rest of the arguments are identified by dependency paths in the style of syntactic rules for events. The anchor mention is specified by giving it a name other than *trigger* and a desired label, e.g., `anchor:Label`.

4.1.1. Predicate-argument Syntactic Paths

The dependency path between a predicate and an argument 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, in Figure 2, the dependency "inhibits" → "ubiquitination" is outgoing ("inhibits" is the head), but it is considered incoming when traversed in the other direction: "ubiquitination" ← "inhibits".

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.

In addition to directionality, dependency patterns support alternation, grouping, and the common regular expression quantifiers. For example, the pattern `nsubj prep_of?` matches exactly one outgoing `nsubj` hop followed by an optional outgoing `prep_of`.

4.1.2. Named Arguments

The arguments in a dependency pattern are written using the `name:label = path` syntax, where `label` is the

[4] Odin also supports explicit rule priorities, which are omitted here for brevity.

[5] Please refer to Appendix C for a detailed Backus-Naur Form (BNF) grammar describing the dependency patterns syntax.

```
Mexico's president was recently reelected.
China 's president was recently reelected.
```

```
1 pattern: |
2   trigger = [lemma=reelect]
3   theme:Entity = nsubjpass (?! poss [lemma=China])
```

Example 3: This dependency pattern contains a negative lookaround to avoid matching mentions referring to the reelection of "**China**'s president" (assuming the domain of interest focuses on the election of Mexican presidents).

label of an existing Odin mention. The path must lead to a token contained in a mention with the specified label.

Odin captures argument arity in events through argument quantifiers. Arguments can be made optional with the `?` operator. The + operator is used to indicate the creation of a single event mention containing all matches for that argument. 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}`. The `ubiq-syn` rule in Example 2 shows an optional `cause` argument.

4.1.3. Token Constraints

Token constraints can be used to restrict a dependency pattern by adding lexical constraints at any point of the path. They are described further in Section 4.2., and a BNF grammar describing their syntax is available in Appendix A.

4.1.4. Lookarounds as Contextual Constraints

Dependency patterns support non-capturing lookaround expressions to constrain syntactic context. The lookaround syntax is `(?= pattern)` for positive assertions and `(?! pattern)` for negative assertions. Example 3 demonstrates a use case for a negative lookaround.

4.2. Surface Patterns

The same fundamental features of dependency patterns are also supported by surface patterns[6], which operate independently of syntax.

Tokens are described using one or more constraints on lexical, morphological, or semantic attributes written in the form `[attribute=value]`. Example 1 includes an optional token preceding the event's theme described in terms of its POS tag ("DT"), rather than a disjunction of possible words that might satisfy the pattern (e.g., "a", "the", etc.). The value of these token attributes may be given as exact strings or regular expressions (e.g. `[lemma=/[eo]r$/]` for all lemmas ending in "er" or "or"). More complex constraints can be expressed using boolean expressions. For example, line 17 of Example 2 shows a conjunction of constraints on the lemma and the POS tag of the same token.

[6]Please refer to Appendix B for a detailed BNF grammar describing the surface patterns syntax.

```
Odin's patterns can be very precise.
This is precisely the point.
We managed to improve PRECISION.
```

```
1 [word=/(?i)^precis/ & !tag=RB]
```

Example 4: A token pattern involving two constraints on a single token. The token must begin with "precis" (`(?i)` indicates that the match is case insensitive) and cannot be an adverb.

4.2.1. Named Arguments

Surface patterns may be used to describe events or relations using named arguments that are created either on-the-fly using the `(?<argname> token sequence)` syntax shown in line 11 of Example 1, or in reference to an existing mention by using the `@argname:Label` syntax shown on line 12 in Example 1.

4.2.2. Lookarounds as Contextual Constraints

Surface pattern may be honed with lookbehind and lookahead expressions that impose constraints on the sentential context of a match. These assertions may be either positive (i.e., the contained pattern must exist) or negative (i.e., the contained pattern must not exist). Lookbehinds use the `(?<= token sequence)` syntax for positive assertions and `(?<! token sequence)` for negative assertions; positive lookaheads use `(?= token sequence)`, while negative lookaheads are specified using `(?! token sequence)`. Notably, Odin supports efficient unrestricted variable length lookbehinds, which is uncommon for regular expression engines (Friedl, 2006). An example of a negative lookbehind is shown in Example 5.

```
1 (?<!China) []? (?<theme>/[pP]resident/)
2 was (?<trigger>[lemma=reelect])
```

Example 5: This surface pattern is analogous to the dependency pattern in Example 3. Here a negative lookbehind is used to avoid matching mentions referring to the reelection of "**China**'s president".

4.3. Multiple Rule Files

Including all the rules in a single file is feasible for small IE systems, but systems targeting larger domains require organizing rules in a way that promotes modularity and rule reuse. Odin supports multi-file grammars, as well as variables that can be used within and across grammar files. When grammars are split in multiple files, Odin reads the top-level file (the master grammar) and imports the other grammars as needed. These imported grammars may themselves import other grammars. Each file can define default values for the variables it uses, and these values can be overridden at import time.

We refer the reader to the "Building a Grammar" section of the manual for a more detailed explanation and further examples of master grammars and template grammars.

```
- Entity:
    - Gene
    - Protein
    - SmallMolecule
- Event:
    - Conversion:
        - Phosphorylation
        - Ubiquitination
        - Hydroxylation
        - Sumoylation
        - Acetylation
    - Control:
        - Catalysis
        - Modulation
```

Example 6: A formalized taxonomy of biochemical entities and interactions inspired by BioPAX (Demir et al., 2010).

4.4. Taxonomy

We have shown in Examples 1 and 2 that rules can assign more than one label to an extracted mention. This allowed us to define an ad-hoc taxonomy that states that a `Ubiquitination` is also an `Event`.

The ability to define ad-hoc taxonomies is useful when developing small IE systems, but this can become cumbersome for larger domains. To address this, Odin also supports a formal taxonomy where the label hierarchy is written as a tree (or forest).

Taxonomies are encoded as lists of YAML dictionaries that display the hierarchy of labels in a clear and readable way. An example of a formal taxonomy is shown in Figure 6.

When an explicit taxonomy is available, a rule can use a single label which will include all of its parent labels implicitly. This enforces consistency in the labels of the resulting mentions, and also catches typographical errors by not allowing the use of labels missing from the taxonomy.

More details about the usage of a formal taxonomy are available in the "Taxonomy" section of the manual.

5. Resources

The entire Odin framework is available as part of the Processors NLP library.[7] Processors is written in Scala, which makes interaction with other languages running on the Java Virtual Machine straightforward. To facilitate the quick start of Odin-based projects, we have also made available an example project that implements a simple system using Odin both in Scala and in Java.[8]

We also provide a web interface for developing and debugging Odin rules. The web UI allows one to inspect each sentence's token attributes and dependency graph using Brat visualizations (Stenetorp et al., 2012).[9] A screenshot of this web interface is shown in Figure 3.

6. Acknowledgments

This work was funded by the DARPA Big Mechanism program under ARO contract W911NF-14-1-0395.

[7] `https://github.com/clulab/processors`

[8] `https://github.com/clulab/odin-examples`

[9] `http://agathon.sista.arizona.edu:8080/odinweb/open`

7. Bibliographical References

Abney, S. (1996). Partial parsing via finite-state cascades. Natural Language Engineering, 2(04):337–344.

Appelt, D. E. and Onyshkevych, B. (1998). The common pattern specification language. In Proc. of the TIPSTER Workshop, pages 23–30.

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

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., and Schneider, N. (2012). Abstract meaning representation (amr) 1.0 specification. In Parsing on Freebase from Question-Answer Pairs. In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing. Seattle: ACL, pages 1533–1544.

Ben-Kiki, O., Evans, C., and Ingerson, B. (2005). Yaml ain't markup language (yaml) version 1.1. yaml. org, Tech. Rep.

Chambers, N., Cer, D., Grenager, T., Hall, D., Kiddon, C., MacCartney, B., De Marneffe, M.-C., Ramage, D., Yeh, E., and Manning, C. D. (2007). Learning alignments and leveraging natural logic. In Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing, pages 165–170. Association for Computational Linguistics.

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

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In EMNLP, pages 740–750.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). A framework and graphical development environment for robust nlp tools and applications. In ACL, pages 168–175.

De Marneffe, M.-C. and Manning, C. D. (2008a). The stanford typed dependencies representation. In Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation, pages 1–8. Association for Computational Linguistics.

de Marneffe, M.-C. and Manning, C. D. (2008b). The Stanford typed dependencies representation. In Proc. of COLING Workshop on Cross-framework and Cross-domain Parser Evaluation.

Demir, E., Cary, M. P., Paley, S., Fukuda, K., Lemer, C., Vastrik, I., Wu, G., D'Eustachio, P., Schaefer, C., Luciano, J., et al. (2010). The biopax community standard for pathway data sharing. Nature biotechnology, 28(9):935–942.

Friedl, J. E., (2006). Mastering regular expressions, pages 133–134. O'Reilly Media, Inc.

Levy, R. and Andrew, G. (2006). Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In Proc. of LREC.

Li, Y., Reiss, F. R., and Chiticariu, L. (2011). Sys-

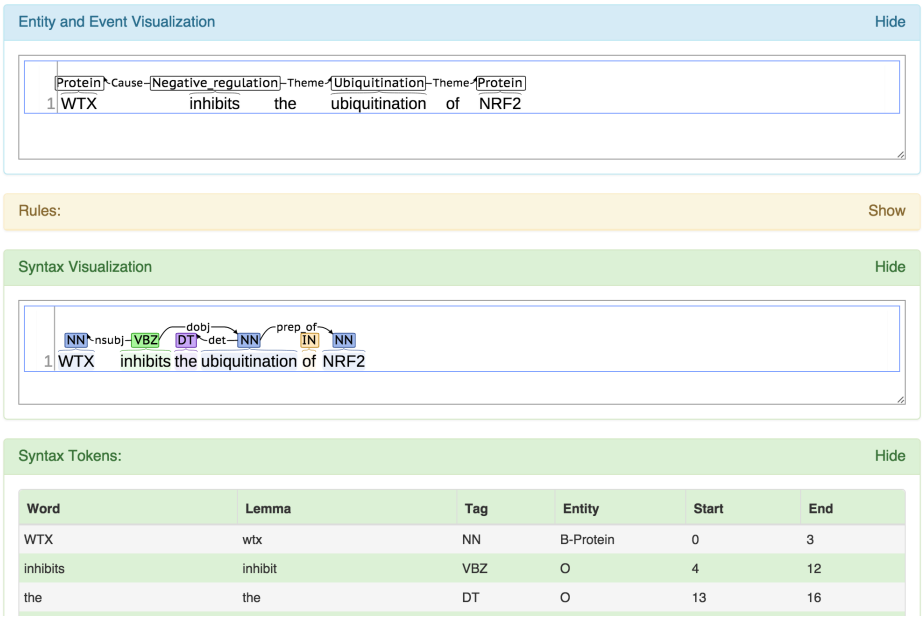

Figure 3: A visualization of Odin's output for the sentence in Figure 1 using the rules in Example 2.

temt: A declarative information extraction system. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Systems Demonstrations, pages 109–114. Association for Computational Linguistics.

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

Piskorski, J., Schäfer, U., and Xu, F. (2004). Shallow processing with unification and typed feature structures–foundations and applications.

Reiss, F., Raghavan, S., Krishnamurthy, R., Zhu, H., and Vaithyanathan, S. (2008). An algebraic approach to rule-based information extraction. In Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, pages 933–942. IEEE.

Stenetorp, P., Pyysalo, S., Topić, G., Ohta, T., Ananiadou, S., and Tsujii, J. (2012). Brat: a web-based tool for nlp-assisted text annotation. In Proc. of the Demonstrations at EACL.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., and Nivre, J. (2008). The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. In Proceedings of the Twelfth Conference on Computational Natural Language Learning, pages 159–177. Association for Computational Linguistics.

Thakker, D., Osman, T., and Lakin, P. (2009). Gate jape grammar tutorial. Nottingham Trent University, UK, Phil Lakin, UK, Version, 1.

Valenzuela-Escárcega, M. A., Hahn-Powell, G., and Surdeanu, M. (2015a). Description of the Odin Event Extraction Framework and Rule Language. ArXiv e-prints, September.

Valenzuela-Escárcega, M. A., Hahn-Powell, G., Hicks, T., and Surdeanu, M. (2015b). A domain-independent rule-based framework for event extraction. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics: Software Demonstrations (ACL).

Appendix A Token Constraint Grammar

The following grammar describes Odin's token constraints. A token constraint is a boolean expression over a token's lexical, morphological, or semantic attributes. The grammar is written in Backus-Naur form (BNF), with terminals enclosed in quotes, non-terminals enclosed in angle brackets, and optional items enclosed in square brackets. The $*$ character is the Kleene star.

⟨*TokenConstraint*⟩ ::= '[' [⟨*DisjunctiveConstraint*⟩] ']'

⟨*DisjunctiveConstraint*⟩ ::= ⟨*ConjunctiveConstraint*⟩
('|' ⟨*ConjunctiveConstraint*⟩)*

⟨*ConjunctiveConstraint*⟩ ::= ⟨*NegatedConstraint*⟩
('&' ⟨*NegatedConstraint*⟩)*

⟨*NegatedConstraint*⟩ ::= ['!'] ⟨*AtomicConstraint*⟩

⟨*AtomicConstraint*⟩ ::= ⟨*FieldConstraint*⟩
| '(' ⟨*DisjunctiveConstraint*⟩ ')'

⟨*FieldConstraint*⟩ ::= ⟨*FieldName*⟩ '=' ⟨*StringMatcher*⟩

⟨*FieldName*⟩ ::= 'word'
| 'lemma'
| 'tag'
| 'entity'
| 'chunk'
| 'incoming'
| 'outgoing'
| 'mention'

⟨*StringMatcher*⟩ ::= ⟨*ExactStringMatcher*⟩
| ⟨*RegexStringMatcher*⟩

⟨*ExactStringMatcher*⟩ ::= ⟨*StringLiteral*⟩

⟨*StringLiteral*⟩ ::= ⟨*identifier*⟩
| ⟨*SingleQuoteString*⟩
| ⟨*DoubleQuoteString*⟩

⟨*RegexStringMatcher*⟩ ::= ⟨*RegexLiteral*⟩

Appendix B Token Pattern Grammar

In this section we describe the BNF grammar for Odin's Runes' token patterns, i.e., the surface rules. Token patterns support several advanced features like lazy and greedy quantifiers, named captures of both mentions and sequences of tokens with the ability to share names among the captures, and zero-width assertions.

⟨*TokenPattern*⟩ ::= ⟨*DisjunctiveTokenPattern*⟩

⟨*DisjunctiveTokenPattern*⟩ ::=
⟨*ConcatenatedTokenPattern*⟩
('|' ⟨*ConcatenatedTokenPattern*⟩)*

⟨*ConcatenatedTokenPattern*⟩ ::=
⟨*QuantifiedTokenPattern*⟩ ⟨*QuantifiedTokenPattern*⟩*

⟨*QuantifiedTokenPattern*⟩ ::= ⟨*AtomicTokenPattern*⟩
| ⟨*RepeatedTokenPattern*⟩
| ⟨*RangeTokenPattern*⟩

⟨*AtomicTokenPattern*⟩ ::= ⟨*SingleTokenPattern*⟩
| ⟨*MentionTokenPattern*⟩
| ⟨*CaptureTokenPattern*⟩
| ⟨*AssertionTokenPattern*⟩
| '(' ⟨*DisjunctiveTokenPattern*⟩ ')'

⟨*RepeatedTokenPattern*⟩ ::= ⟨*AtomicTokenPattern*⟩
('??' | '*?' | '+?' | '?' | '*' | '+')

⟨*RangeTokenPattern*⟩ ::= ⟨*AtomicTokenPattern*⟩
'{' ⟨*number*⟩ ',' ⟨*number*⟩ '}'
| ⟨*AtomicTokenPattern*⟩ '{' ⟨*number*⟩ ',' '}'
| ⟨*AtomicTokenPattern*⟩ '{' ',' ⟨*number*⟩ '}'
| ⟨*AtomicTokenPattern*⟩ '{' ⟨*number*⟩ '}'

⟨*SingleTokenPattern*⟩ ::= ⟨*StringMatcher*⟩
| ⟨*TokenConstraint*⟩

⟨*MentionTokenPattern*⟩ ::=
'@' [⟨*StringLiteral*⟩ ':'] ⟨*ExactStringMatcher*⟩

⟨*CaptureTokenPattern*⟩ ::=
'(?<' ⟨*identifier*⟩ '>' ⟨*DisjunctiveTokenPattern*⟩ ')'

⟨*AssertionTokenPattern*⟩ ::=
⟨*SentenceStartTokenAssertion*⟩
| ⟨*SentenceEndTokenAssertion*⟩
| ⟨*LookaheadTokenAssertion*⟩
| ⟨*LookbehindTokenAssertion*⟩

⟨*SentenceStartTokenAssertion*⟩ ::= '^'

⟨*SentenceEndTokenAssertion*⟩ ::= '$'

⟨*LookaheadTokenAssertion*⟩ ::=
('(?=' | '(?!') ⟨*DisjunctiveTokenPattern*⟩ ')'

⟨*LookbehindTokenAssertion*⟩ ::=
('(?<=' | '(?<!') ⟨*DisjunctiveTokenPattern*⟩ ')'

Appendix C Dependency Pattern Grammar

This BNF grammar describes the syntax for Odin's Runes' dependency patterns. These patterns are applied over a dependency graph. Notable features include the usual regex quantifiers (although there is no lazy/greedy distinction), lookaround assertions (again, no distinction between lookahead and lookbehind), and they can pack/unpack arguments using argument quantifiers as explained in Section 4.1.2. Token constraints are also supported as a way of adding lexical constraints at any step of the path.

⟨*DependencyPattern*⟩ ::=
⟨*TriggerPatternDependencyPattern*⟩
| ⟨*TriggerMentionDependencyPattern*⟩

⟨*TriggerPatternDependencyPattern*⟩ ::=
'trigger' '=' ⟨*TokenPattern*⟩ ⟨*ArgPattern*⟩+

⟨*TriggerMentionDependencyPattern*⟩ ::=
⟨*identifier*⟩ ':' ⟨*identifier*⟩ ⟨*ArgPattern*⟩+

⟨*ArgPattern*⟩ ::=
⟨*identifier*⟩ ':' ⟨*identifier*⟩
['*' | '+' | '?' | '{' ⟨*number*⟩ '}'] '='
⟨*DisjunctiveDependencyPattern*⟩

⟨*DisjunctiveDependencyPattern*⟩ ::=
⟨*ConcatenatedDependencyPattern*⟩
(‘|’ ⟨*ConcatenatedDependencyPattern*⟩)*

⟨*ConcatenatedDependencyPattern*⟩ ::=
⟨*StepDependencyPattern*⟩ ⟨*StepDependencyPattern*⟩*

⟨*StepDependencyPattern*⟩ ::= ⟨*FilterDependencyPattern*⟩
| ⟨*TraversalDependencyPattern*⟩

⟨*FilterDependencyPattern*⟩ ::= ⟨*TokenConstraint*⟩

⟨*TraversalDependencyPattern*⟩ ::=
⟨*AtomicDependencyPattern*⟩
| ⟨*RangeDependencyPattern*⟩
| ⟨*QuantifiedDependencyPattern*⟩

⟨*QuantifiedDependencyPattern*⟩ ::=
⟨*AtomicDependencyPattern*⟩ (‘?’ | ‘*’ | ‘+’)

⟨*RangeDependencyPattern*⟩ ::=
⟨*AtomicDependencyPattern*⟩
‘{’ ⟨*number*⟩ ‘,’ ⟨*number*⟩ ‘}’
| ⟨*AtomicDependencyPattern*⟩ ‘{’ ‘,’ ⟨*number*⟩ ‘}’
| ⟨*AtomicDependencyPattern*⟩ ‘{’ ⟨*number*⟩ ‘,’ ‘}’
| ⟨*AtomicDependencyPattern*⟩ ‘{’ ⟨*number*⟩ ‘}’

⟨*LookaroundDependencyPattern*⟩ ::=
(‘(?=’ | ‘(?!’) ⟨*DisjunctiveDependencyPattern*⟩ ‘)’

⟨*AtomicDependencyPattern*⟩ ::=
⟨*OutgoingDependencyPattern*⟩
| ⟨*IncomingDependencyPattern*⟩
| ⟨*LookaroundDependencyPattern*⟩
| ‘(’ ⟨*DisjunctiveDependencyPattern*⟩ ‘)’

⟨*OutgoingDependencyPattern*⟩ ::= ‘>>’
| [‘>’] ⟨*StringMatcher*⟩

⟨*IncomingDependencyPattern*⟩ ::= ‘<<’
| ‘<’ ⟨*StringMatcher*⟩