# Design Document

Philippe Solodov

May 2021

## 1 Abstract

Program verification provides powerful tools for reasoning about and proving the properties of a particular software system. However, program verification tools typically require additional effort or unwieldy syntax which discourages their use in many environments. In addition, when software systems are already documented, the added utility of program verification tools may not justify the added effort.

We use natural language processing (NLP) to automatically analyze the documentation within the source code for a software system and produce specifications wherever possible.

We demonstrate the usage of the tool on programs written in the Rust programming language, using documentation samples from the Rust standard library, and show various stages of analysis, including generation of programmatic specifications from natural language specifications and a keyword search mechanism to find relevant functions.

## 2 Introduction

Program verification refers to the process of generating a formal proof of a program's behaviour, allowing guarantees to be made about the properties of a program's output or behaviour for a specified set of inputs. Deductive program verification is a subset of program verification, where automatic tools generate complete proofs of a program's behaviour and output from a partial user specification, allowing verification to occur with significantly less effort [Filliâtre 2011]. Program verification is useful for compiling and optimizing programs, as being able to prove that a program does not execute specific statements or functions means that less code can be generated. It is also useful for verifying that critical software systems do not demonstrate unexpected behaviour - for instance, the software that runs life-support systems or flight systems in planes should avoid crashing or reporting incorrect information wherever possible.

Rust is a compiled language, designed to prioritize safe programming, utilizing high level abstractions, while simultaneously achieving performance similar

to that of popular systems programming languages such as C and C++. In particular, it provides both functional and imperative styles of programming, which is made especially effective with Rust's ability to specify the mutability of every variable defined in a Rust program. This allows Rust to ensure that variables do not have more than one mutable reference at a time, which enables a number of optimizations and safety guarantees. These guarantees allow equivalent Rust code to outperform code written in other languages, per the results from [*Which programming language is fastest?*]. It also simplifies verifying Rust programs, as verifiers do not need to account for the possibility of variables being mutated in more than one place at a particular time [Astrauskas et al. 2019].

The lack of mutable aliases and ability to specify the mutability of function arguments makes Rust's type system more robust, as implementors can be confident that if their implementations are correct, type invariants will not be violated by user error. To make these invariants even stronger, Rust users can use Prusti, which is a deductive program verification tool for the Rust programming language [Astrauskas et al. 2019]. Users can provide specifications using Prusti's syntax, consisting of preconditions and postconditions with boolean statements, and including universal quantifiers. Preconditions are specified using the `requires` keyword, and postconditions are specified with the `ensures` keyword. Universal quantifiers can be defined with the `forall` keyword. An example of a specification using these keywords can be seen in Fig 1.

Prusti can then automatically verify these specifications by building proofs. This also extends to proving that a program or function does not crash (or in Rust terminology, panic), given some set of specifications. This greatly simplifies the process of program verification, as formal proofs are generated by Prusti itself, with no user intervention required. Prusti performs this verification by transforming annotated Rust programs into equivalent programs in the Viper language, which are then verified through Viper's toolchain.

Viper is a system for program verification, which provides functionality for processing the intermediate Viper language into input for a first-order logic solver to generate proofs of a program's behaviour. Viper is very flexible - it provides different backends for different styles of verification, and due to its human-readable nature, new program verification techniques can be written in Viper itself, instead of requiring bespoke implementations. This flexibility, and the functionality provided through the Viper system, greatly reduces the effort required to build program verifiers for new languages through a simpler translation to Viper itself [Müller, Schwerhoff, and Summers 2016].

The goal is to build a system which enables transforming documentation into the specifications that Prusti uses, which hopefully reduces the work required to use program verification tools, by allowing programmers to either use preexisting documentation, or to extend existing documentation and use Prusti specifications as is convenient. Additionally, this system should be interpretable, and not overly restrictive in the style of writing it takes. Having to learn why an opaque system transforms text the way it does, or the style of writing that the system prefers, would make the explicit Prusti specifications vastly preferable, as users would only have to learn a few additional syntax elements. Such a sys-

tem could also help users ensure that their documentation remains consistent with the behaviour of their programs, even when modifications occur.

# 3 Background

**Natural Language Processing** Natural language processing (NLP) refers to the process of automatically analyzing and representing human language. Current methods use algorithmic, statistical, and machine learning approaches to determine the sentiment, structure, topics, and facts contained within a particular portion of text, and do so at different levels of granularity [Cambria and White 2014].

**Tokenization** Tokenization in the context of NLP refers to the process of splitting text into tokens, or smaller pieces of text. For English text, splitting typically occurs at the level of punctuation and white space, but can also split hyphenated words into two pieces, or words with a possessive apostrophe into two tokens. Depending on the domain from which the text originates, tokens can also include special cases, such as whole links or email addresses [Manning, Raghavan, and Schütze 2008].

**Part of Speech Tagging** A part-of-speech (POS) describes what role a particular word or token has in a sentence. POS tags are generally grammatical components of a sentence, and typical examples of POS tags include nouns, verbs, adjectives, adverbs, and punctuation, though this is not an exhaustive list. Most models also have more specific tags, incorporating elements such as the tense of the word into the tag. A variety of methods exist to generate these POS tags [Mitkov 2005].

**Context-Free Grammars** Context-free grammars (CFG) describe a set of production rules. Production rules map a symbol to one or more other symbols, or terminate. In the case of natural language processing, these production rules would terminate in a token generated through tokenization. Probabilistic context-free grammars (PCFG) are an extension of CFGs, which assign a probability to each production rule, allowing for the likeliest parse-trees to be generated. These parse-trees are useful as input to other methods, which can discover features of the text, such as named entities or keywords. The quality of the parse-trees depends on the quality of the underlying production rules and the dataset from which the probabilities of each particular production rule are derived, if such a dataset is used [Pullum and Gazdar 1987].

**Named Entity Recognition** Named entity recognition refers to the process of detecting important entities within text. These entities can include dates, people, places, or other domain-specific features which are relevant. Methods can incorporate rule-based systems - for instance, matching any string which is a hyperlink, or matching a particular set of keywords, also referred to as

```
impl PrustiVec {
    /// Specifications elided.
    fn remove(&mut self, index: usize) -> u8 {
        unimplemented!()
    }
    /// Removes the last element from a vector and returns it, or
↪   `None` if it
    /// is empty.
    /// Requires that the list contains items to pop
    #[requires(self.len() >= 1)]
    /// Ensures that the list will have one less item after pop
↪   is called.
    #[ensures(self.len() == old(self.len()) - 1)]
    #[ensures(self.len() >= 0)]
    /// Ensures that the last item of the list is returned.
    #[ensures(result == old({
        let l = self.len() - 1;
        self.lookup(l)
    }))]
    /// Checks that all remaining items remain unmodified.
    #[ensures(
        forall(|i: usize|
            (0 <= i && i < old(self.len()) - 1) ==>
↪   self.lookup(i) == old(self.lookup(i))
        )
    )]
    fn pop(&mut self) -> u8 {
        let l = self.len() - 1;
        self.remove(l)
    }
}
```

Figure 1: Example of a specification for the pop operation on a `Vec` containing `u8` elements. Lines prefixed with a triple forward slash document the specification annotations prefixed with the "#" symbol.

gazetting. Other information, such as POS tags and the structure of the text can also be used to determine which tokens in a text are important [Cambria and White 2014].

**Semantic Role Labelling** Semantic role labelling (SRL) is used to determine what role entities perform in a particular text, relative to a predicate within the text. In the sentence "Person A does something to Person B", SRL would be responsible for determining that "Person A" is an individual, who "does something" to "Person B", the recipient [Màrquez et al. 2008].

**Word Vectors** Natural language can be modelled as a vector space with a large number of dimensions, and words in a particular language can be modelled as vectors within that space. In this space, words which are similar will have corresponding vectors which are also similar - so "cat" and "kitten", which are similar words, would also be very similar in the word vector space. Recent models also aim to ensure that the differences between two word vectors capture information about their relationship. For instance, in natural language, a kitten is to a cat, what a puppy is to a dog. In word vector representation, this would be equivalent saying that $cat - kitten = dog - puppy$ [Pennington, Socher, and Manning 2014]

# 4 Related Works

We discuss here work related to ours, and ways in which our work expands/is different from upon the functionality of these in various ways.

## 4.1 Combining Formal and Machine Learning Techniques for the Generation of JML Specifications

The authors implement a system for generating annotations for source code, and provide a mechanism to assist users in understanding source code by highlighting important elements of documentation, using NER and SRL. The paper provides a mechanism for annotation functions, and these annotations provide information about whether a function is pure, with no side effects, whether it returns nullable values, and whether the function modifies any items [Puccetti, Chalendar, and Gibello 2021]. Our paper incorporates the paper's NER and SRL models, making it feature equivalent in highlighting important elements of documentation. However, our paper can also create programmatic specifications from natural language specifications within the documentation, and users also can perform natural language queries to find relevant functions when specifying their programs.

### 4.2 Automatic Requirements Specification Extraction from Natural Language

The authors implement a system for analyzing a natural language specifications to produce a series of type rules and predicates, which allows capturing information about state transitions and inviolable constraints [Ghosh et al. 2016]. Our paper does not perform any analysis of type-level rules, as Rust provides a robust type system which can capture many of these type-level rules, and specifications at the function level can describe any type transitions or invariants that are upheld during execution. The authors also make use of StanfordNLP's dependency parser [Manning et al. 2014], which allows establishing the relationship between words in a text. Our paper does not make use of any dependency information, which makes certain types of analysis, such as determining the subject performing an action or the subject being modified by an action, significantly more difficult.

## 5 Method

In this section, we describe the general methods used in the system to produce programmatic specifications from natural language specifications, as well as the major steps in producing these specifications. We display the output of the various stages of the process on the function `u32::is_power_of_two`, as seen in Fig 2, which was specifically chosen as it demonstrates many of the major components of the system. In particular, it is used to demonstrate tokenization and retokenization, POS tagging, and how the CFG is used to create a parse-tree.

```
/// Returns `true` if and only if `self == 2^k` for some `k`.
fn is_power_of_two(&mut self, val: u8);
```

Figure 2: Documentation from `u32::is_power_of_two` [*Primitive type u32*] in Rust's standard library, which demonstrates usage of code blocks, and an existential quantifier.

When referring to documentation here, and elsewhere in this paper, we specifically refer to any text in block comments preceding the type signature of a function. In Fig 3, we see the documentation for `i32::rotate_right`, which contains multiple sentences. When referring to natural language specifications, we refer to sentences such as the first documentation sentence for `i32::rotate_right`. In contrast, the second documentation sentence is an example of a sentence which is not a specification. In general, documentation may contain some number of natural language specifications, but this is not guaranteed, and a particular function may simply have no documentation. The natural language specifications in documentation may also not fully specify the

behaviour of the function. Fig 1 provides examples of programmatic specifications, appearing in the annotations, next to natural language specifications in the documentation.

```
    /// Shifts the bits to the right by a specified amount, n,
↪   wrapping the truncated bits to the beginning of the resulting
↪   integer.
    ///
    /// Please note this isn't the same operation as the >>
↪   shifting operator!
    pub const fn rotate_right(self, n: u32) -> i32;
```

Figure 3: Documentation from `i32::rotate_right` [*Primitive type i32*] in Rust's standard library. The first documentation sentence is an example of a natural language specification, while the second sentence is not.

Rust documentation can also contain subsections, which are separated by headers, as seen in Fig 4. A header is a line beginning with the "#" symbol. When creating programmatic specifications for a function, we only consider sentences from the main section of documentation, which appears before any subsection. Subsections are generally used to demonstrate examples of how the function is used, or provide other details about the function, such as why it exists.

```
/// Main documentation. Sentence 2.
///
/// # Subsection 1
/// ...
fn example() {
    unimplemented!()
}
```

Figure 4: Code sample, demonstrating subsections in Rust documentation. Subsections begin with a line prefixed with one or more occurrences of the "#" symbol.

## 5.1 Overview

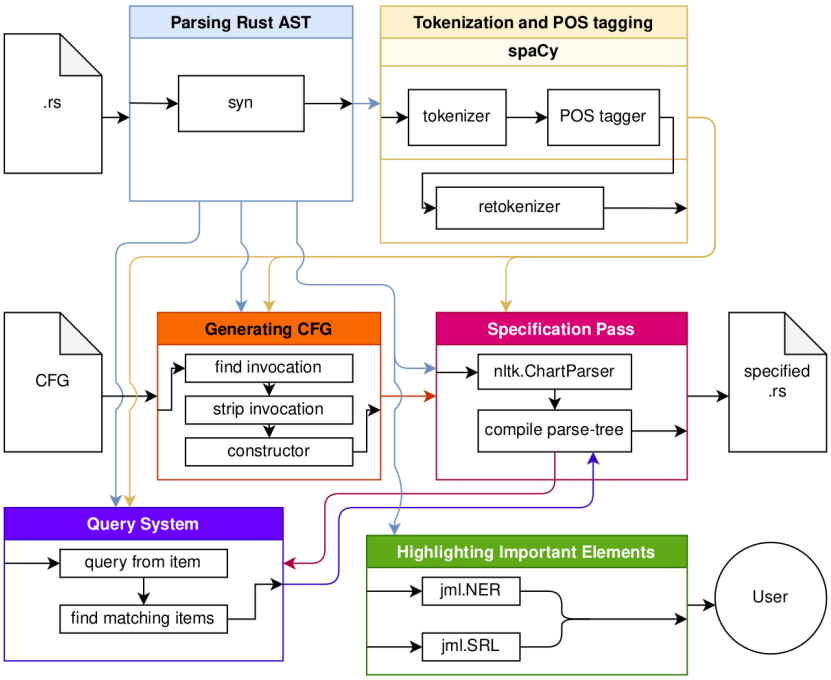


Figure 5: High level structure of the system. Rust files are provided as input, and Rust files with annotations are produced as output. The major components of this system are discussed in the following sections. jml in Highlight Important Elements refers to [Puccetti, Chalendar, and Gibello 2021], which provides the NER and SRL models used in the module.

At a high level, the system takes as input a Rust source file and a CFG we built for this task. The Rust file is parsed into an abstract syntax tree (AST). The documentation within this AST is then tokenized, and POS tags are generated, and are used together in all other phases, with the exception of the highlighting phase. In the CFG, query system, and specification pass, the documentation and function signatures are used from the AST. The specification pass attaches the generated specifications as annotations to the AST, which is converted back into a specified Rust source file. The output file contains the skeleton structure of the input file, which can be used to verify the input source file using Prusti. Alternatively, the user can copy relevant specifications to their source file manually, and then verify it with Prusti.

In addition to specific, the system can also perform natural language queries over the AST, and can also highlight important entities in the text. The output

of these operations is displayed to the user.

## 5.2 Parsing Rust Abstract Syntax Tree

To discover documentation for functions and methods, as well as other information that can be used during the process of type detection, it is first necessary to build a model of the entire program we want to specify. This is accomplished by building the abstract syntax tree of the input program, and then traversing the AST to extract all documentation, functions, types, constants, and other relevant information which could be referenced in a specification. For Rust programs, we use Rust's syn library, which allows building an AST from any input Rust program.

## 5.3 Tokenization and Part-of-Speech Tagging

In the initial parser pass, we tokenize and perform POS tagging on all sentences in relevant portions of the documentation. This tokenization and tagging process is useful for the custom-built CFG used in this project, which supports structured text, with conditional statements, predicates, nested and/or repeated objects.

We use the pretrained "en_core_web_lg" neural network model from the "spaCy" Python library [Honnibal et al. 2020], which performs both tokenization and POS tagging. It is both accurate and fast, which in turn makes the specification process significantly faster. However, as the model was not trained on this domain, it does make systematic errors in tagging, which require manual correction, and the tokens it produces for documentation have undesirable properties. "spaCy" provides pattern matching tools for the tokens produced by the model, which makes correct tagging errors and fixing tokenization issues, related to code blocks and strings in Rust documentation, very simple. In Fig 6, we can see the default output of the "spaCy" tokenizer.


Figure 6: Tokens with associated POS tags for documentation from `u32::is_power_of_two`, using default spaCy tokenizer. Visualization generated using displaCy. Individual blocks contain the underlying text, followed by the POS tag in bold. Blocks are color-coded according to broad roles - blocks in green are nouns and adjectives, and blocks in grey are punctuation and related symbols. Information about the POS tags that spaCy produces can be found in Appendix A.

We can see that the tokenization process splits the code blocks `true` and `self == 2^k` into multiple tokens. These should not be split, as these are

considered singular elements in the CFG. Likewise, documentation may contain other numeric or string literals, which are also not tokenized appropriately. These tokenizations errors are corrected using spaCy's pattern matching functionality. Tokens within quotes are merged, as these are considered to be string literals. Additionally, to improve the specificity of the parse-trees generated, domain-specific POS tags are applied to words such as "if" and "returns", using the same pattern matching functionality. We can see the resulting tokens and POS tags after all corrections have been applied in Fig 7. A number of other transformations are also applied, which serve to make the token stream compatible with the CFG created for this project, but do not appear in the example below. A complete list of custom POS tags is available in Appendix A.

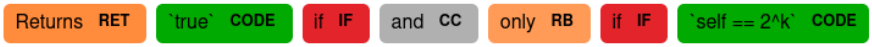


Figure 7: Tokens with associated POS tags for `u32::is_power_of_two`, as in Fig 2, using default spaCy tokenizer and task-specific retokenization procedure. Visualization generated using displaCy.

## 5.4 Generating Context-Free Grammar

We use a task-specific CFG, which maps the tokens and associated POS tags produced in Section 5.3 to a tree structure. We selected production rules by inspecting common patterns in programmatic and natural language specifications, including syntactic elements. A few of the rules that have been produced include rules for true/false expressions, such as boolean literals and predicates, if statements, consisting of the word "if" followed by a true/false expression, and return statements, consisting of the word "return" followed by some object. In Fig 8, we see an example of the OBJ production rule, which detects objects in text. A comprehensive list of rules is included in Appendix B.

However, this CFG is not exhaustive, and does not cover all of the ways in which a user could refer to invoking a particular function in their documentation. To address this, it is possible to find information about how a function is invoked from the documentation of that function. The first sentence of the main section of Rust documentation is typically structured in a "Returns the result from some operation" format, and can be used as a template for a production rule.

To transform this sentence to a production rule, we first tokenize it and assign the POS tags to the tokens, using "spaCy". Leading keywords, such as "returns", and determinants, such as "the" are then discarded when creating the production rule, as these will not necessarily appear when a user invokes the function, and are covered by other production rules. A constructor is then created, which will parse any text which is matched by the production rule, and map input items to their correct location in the function input, according to both their type and location in the text. This production rule is added to the grammar, and if a sequence of tokens matching the rule is seen in the future, it

```
# Objects
OBJ → PRP | (DT)? MNN | CODE | (DT)? LIT
| OBJ OP OBJ | (DT)? FNCALL | DT VBG MNN | PROP_OF OBJ
```

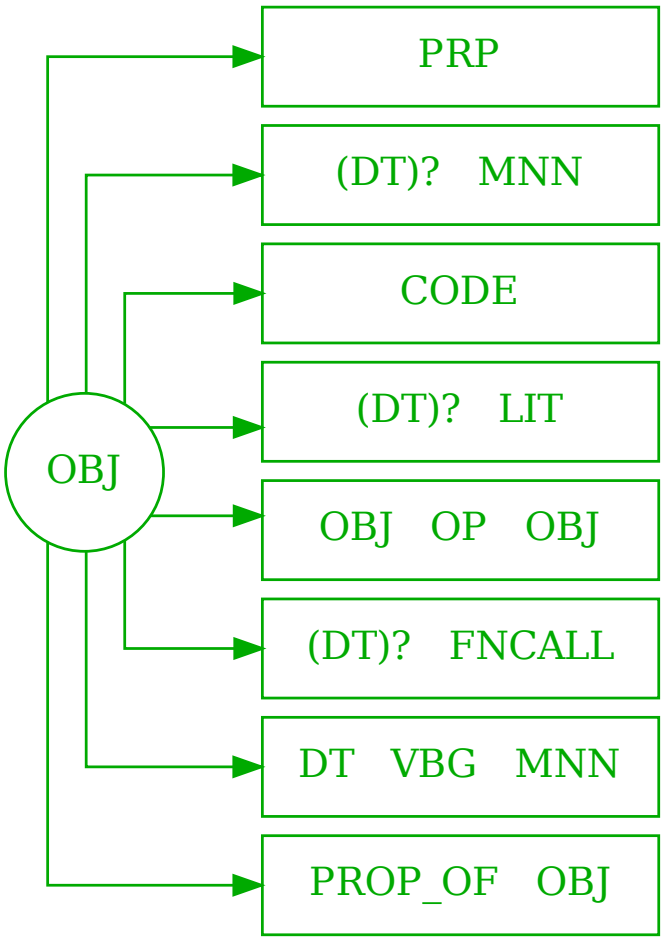


Figure 8: An example of the OBJ production rule, in CFG syntax and in tree form. Within individual box is a space separated list of possible groups of POS tags and/or CFG production rules that OBJ maps to. FNCALL refers to a function call, for which production rules are generated in this stage.

will be matched to that specific function, allowing it to be used as a component of future natural language specifications.

Once the base CFG is updated with the project-specific production rules, it now contains a set of base production rules which capture many of the syntactic elements of program documentation and natural language specifications, and a number of custom rules for functions in the program being specified.

## 5.5 Specification Pass

During the specification pass, functions to be specified are identified, and then natural language specifications are transformed into programmatic specifications and applied to the input program, wherever possible.

This is accomplished by traversing the functions within the AST from Section 5.2. As some functions may already have specifications, or do not need to be specified, any function that is intended to be specified must be annotated with `#[specify]`.

We use the nltk library [Loper and Bird 2002] to produce the parse-tree for a sentence from our CFG. nltk provides the ChartParser class, which can create parse-trees for CFGs containing self-referential production rules. We use self-referential rules to parse lists with some number of repeated items (eg. an operation over multiple items), or text which has nested objects (e.g. a function call which references the output of another function call). Other libraries were not evaluated, as nltk provided all necessary functionality out of the box and is sufficiently performant.

In Fig 9, we see the parse tree resulting from the tokens generated for `u32::is_power_of_two`. The leaf nodes and their direct parents are the tokens and their POS tags, respectively. All remaining nodes correspond to production rules. A comprehensive list of production rules is provided in Appendix B.

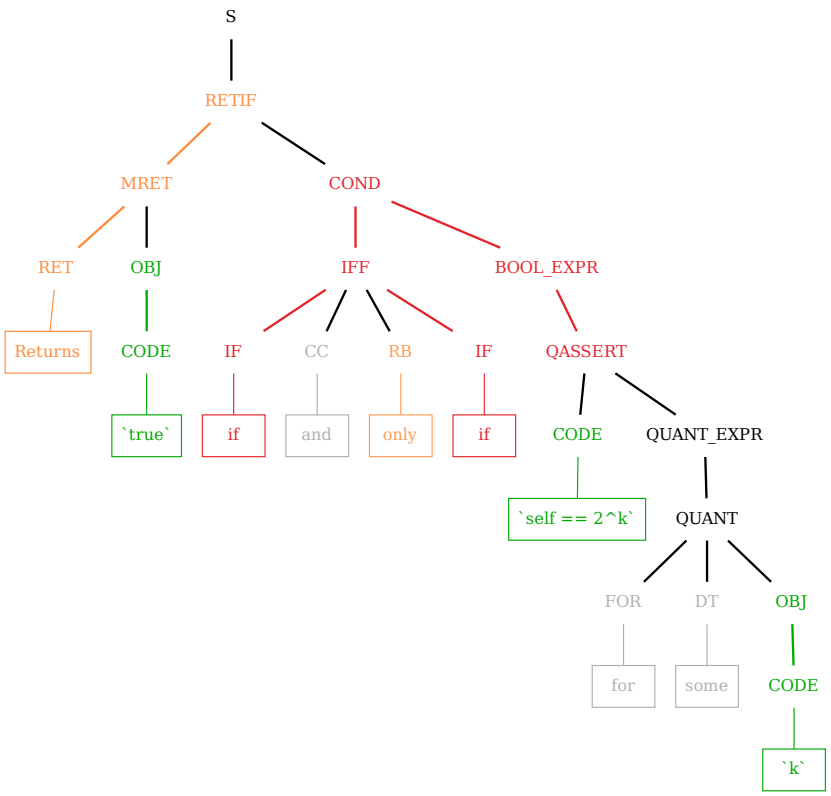


Figure 9: Resulting parse tree for `u32::is_power_of_two` from Fig 2, generated by nltk.ChartParser, using tokenation and POS tags from Section 5.3, and CFG from Section 5.4.

This grammar is then compiled into a Prusti specification in Fig 10. This process is largely a one-to-one transformation of syntactic production rules into their Rust equivalents, and other production rules into an approximate Rust-appropriate representation. Prusti does not support existential quantifiers, represented by the "forsome" function, so this specification would not be successfully verified, but a similar function using the quantifying statement "for all `x`", could be verified, and the process would otherwise be the same. We note that the value `2^k`, which in the context of `is_power_of_two` represents "2 to the power of k", is interpreted as "2 xor k" in Rust syntax. We use code blocks within documentation as an escape-hatch for users, who might find that explaining a particular operation in natural language is difficult or overly verbose, or want to partially bypass the natural language analysis methods of this project. Accordingly, we do not attempt to interpret the code contained with a code

block in any way, and assume that the user's code is correct.

```
/// Returns `true` if and only if `self == 2^k` for some `k`.
#[ensures(forsome(|k: int| self == 2^k && result))]
#[ensures(forsome(|k: int| result ==> self == 2^k))]
fn is_power_of_two(&mut self, val: u8);
```

Figure 10: Fully specified u32::is_power_of_two. Minor stylistic alterations have been made for consistency. The"forsome" function does not exist in Prusti, and Prusti does not support the existential quantifier in this function's documentation.

## 5.6 Important Element Highlighting

In cases where generating specifications is not possible, or the specification system fails, providing feedback about the important elements of the documentation makes it easier for users to determine which elements of the documentation are relevant when building their own specification.

Named-entity recognition (NER) and semantic-role labelling (SRL) are two natural language processing techniques which can be used to detect functions and items in a text. NER, as its name suggests, extracts named entities from a text, which in this problem domain, are functions, function inputs, and function outputs. SRL is further used to determine the relationship between entities and the actions they perform, which can be used to determine which elements of a function signature they correspond to. We use the NER and SRL models from [Puccetti, Chalendar, and Gibello 2021], which are trained on program documentation.

```
/// Removes the last element from a vector and returns it, or
↪ `None` if it is empty.
pub fn pop(&mut self, index: usize);
```

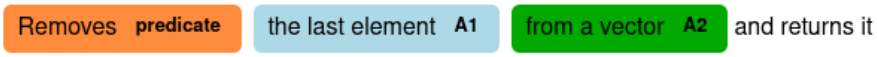


Figure 11: Documentation, and resulting SRL annotations for Vec::pop. The NER model did not find any entities in this text. NER/SRL models from [Puccetti, Chalendar, and Gibello 2021] were used.

In Fig 11, the SRL model annotates the first section of the first sentence from Vec<T>::pop. It detects the predicate "Removes", and that the vector and last element are arguments, though it does not find the "and returns it" predicate at the end of the sentence. Overall, the SRL tool captures the essential components

in the sentence. A user could use this information to very quickly determine that `Vec::pop` can be specified in terms of `Vec::remove`, and manually copy any relevant specifications from the `Vec::remove` function. To assist in the discovery of related functions, the user can also use the query system, which is discussed in Section 5.7.

## 5.7 Query System

The query system provides a mechanism to discover items in a program which contain specific textual elements, and have a particular type signature. This is useful both for users, who may want to find a function that takes a particular type as input or has uses a particular phrase in its documentation, and for the specification process, which may need to discover what function or item is being referred to in a particular portion of documentation. In Fig 12, we can see that `fn overflowing_neg(self) -> (i32, bool)` references the negate operation, implemented by `fn neg(self) -> i32`.

```
/// Performs the unary `-` operation.
fn neg(self) -> i32;

/// Negates self, overflowing if this is equal to the minimum
↪  value.
fn overflowing_neg(self) -> (i32, bool);
```

Figure 12: Documentation from `i32::neg` and `i32::overflowing_neg` [*Primitive type i32*] in Rust's standard library, demonstrating a reference to the negation operation, elsewhere in the code. All but the first sentence of the documentation was omitted for emphasis.

To discover all of the items matching a particular query, it is necessary to search the AST for functions with matching or similar documentation and with appropriate type signatures. As users may not always use the exact words that appear in the function documentation, the matching procedure is flexible, and allows different sentence formats, usage of synonyms, and does not require all of the words in the text of interest to appear in the documentation for a particular function.

From Fig 13, we can see that the we see that `i32::MIN` and `fn min` use the keywords "smallest" and "minimum" respectively, but the user could refer to the "smallest of two values", or "the minimum value of i32". The flexible matching procedures should return both `i32::MIN` and `fn min` in either case, as "smallest" and "minimum" are interchangeable in this case. To rectify this, functions should also be ranked based on their similarity to the text of interest. A number of heuristic methods exist, including spaCy's sentence similarity functions, which use the distance between word vector representations of the sentences.

A more appropriate method may also incorporate structural information about the documentation into the analysis, as we can see that `i32::MIN` and `fn min` have different syntactic structure, which is also likely to appear when they are invoked. Such a mechanism is present in the CFG, which matches functions with an identical syntactic structure - though this is overly strict in many cases.

```
/// The smallest value that can be represented by this integer
↪  type.
pub const MIN: i8 = -128i8;
```

```
/// Compares and returns the minimum of two values.
/// ... (text elided)
pub fn min<T: Ord>(v1: T, v2: T) -> T {
    v1.min(v2)
}
```

Figure 13: Documentation from `i32::MIN` [*Primitive type i32*] and `fn min<T>` in Rust's standard library, respectively. These declarations demonstrate overlaps in terminology.

It is also possible to refer to a particular function in two or more ways. In Fig 14, we see that `i32::pow` performs the "power" operation, which is the same as the "exponentiation" operation. Using either keyword should return `i32::pow` as possible candidate, and a separate similarity metric should be computed for both keywords when choosing the most similar function.

```
/// Raises `self` to the power of `exp`, using exponentiation by
↪  squaring.
pub const fn pow(self, exp: u32) -> i32;
```

Figure 14: Documentation from `i32::pow` [*Primitive type i32*] in Rust's standard library, which contains multiple keywords that describe the power operation.

Fig 16 shows the functions returned by the query system for the string "The minimum of two values". We can see that the location of the words within the documentation is not important, and that the query system is capable of matching sentences that only partially match the query.

```rust
/// Minimum with the current value.
/// Finds the minimum of the current value and the argument
↪  `val`, and
/// sets the new value to the result.
/// Returns the previous value.
/// ...
fn fetch_min(&self, val: u8, order: Ordering) -> u8;

/// Returns a reference to the first value in the set, if any.
/// This value is always the minimum of all values in the set.
/// ...
fn first(&self) -> Option<&T>;

/// Compares and returns the minimum of two values.
/// Returns the first argument if the comparison determines them
↪  to be equal.
/// ...
#[must_use]
fn min<T: Ord>(v1: T, v2: T) -> T;

/// Returns the minimum of two values with respect to the
↪  specified comparison function.
/// Returns the first argument if the comparison determines them
↪  to be equal.
/// ...
#[must_use]
fn min_by<T, F: FnOnce(&T, &T) -> Ordering>(v1: T, v2: T,
↪  compare: F) -> T;

/// Returns the minimum of two `f32` values.
/// ...
fn minnumf32(x: f32, y: f32) -> f32;

/// Returns the minimum of two `f64` values.
/// ...
fn minnumf64(x: f64, y: f64) -> f64;
```

Figure 15: The output generated by the query system for the query "The minimum of two values", in no particular order. Duplicate functions have been removed, and sections of documentation are elided with ellipses for emphasis. `fetch_min` is implemented for most numerical types, so only a single instance is kept.

The query system is also capable of matching functions with appropriate

type signatures. Using the same query string as in Fig 16, but also specifying that the output is of type `f32`, returns only `minnumf32`.

```
/// Returns the minimum of two `f32` values.
/// ...
fn minnumf32(x: f32, y: f32) -> f32;
```

Figure 16: The output generated by the query system for the query string "The minimum of two values", and output type `f32`. Duplicate functions have been removed, and sections of documentation are elided with ellipses for emphasis.

# 6 Implementation

This project is largely implemented in Python in order to make use of a variety of natural language processing libraries, including spaCy and nltk. We also use specifically built Python bindings for syn, which handles parsing of the AST.

This project uses a number of tools which have not been previously mentioned in this paper. These include visualization generation of parse-trees, Rust ASTs, POS tags, and entity tags, some of which appear throughout this paper. Additionally, in order to produce specifications for projects with complex compilation procedures or macros, a tool is provided to parse the HTML documentation pages created by Cargo, Rust's dependency manager and build system. Cargo's output includes documentation which may otherwise be inaccessible to the AST produced by syn, and parsing this instead of the Rust source file can allow more thorough specification of a particular project. In particular, we use this tool is used to test the project against the Rust standard library, an analysis of which is seen in Section 7.

The source code is available at https://github.com/philippeitis/nlp_specifier/. Installation and usage instructions are provided at https://github.com/philippeitis/nlp_specifier/blob/master/README.md, which details usage of scripts to download certain dependencies and models, and also specifies dependencies that must be manually installed.

# 7 Evaluation and Limitations

It is difficult to produce quantitative analysis for this problem, as there is no existing dataset of ground-truth specifications for code. This means there is no way to determine whether a particular specification for a function is correct without checking manually.

We analyze the performance of this project in three ways:

1. Determining how many sentences of documentation can actually be parsed into a tree with our CFG.

2. Determining the quality of generated specifications in several examples, which highlight strengths and limitations of the system.

3. Determining the reasons why a particular sentence can not be parsed into a tree with our CFG.

## 7.1 Analyzing Generated Specifications

We measure the performance of this project against 3586 unique sentences in the public function documentation in the Rust standard library. The system generated 382 parse trees for only 212 of these unique sentences. As the CFG recognizes more features than the system can currently turn into specifications, only 125 of these parse-trees were successfully converted into specifications, corresponding to 96 sentences.

Of 100 randomly inspected sentences with parse-trees generated, 38 had specifications. Of these 38 sentences, 30 had programmatic specifications which successfully captured the structure of the sentence. In 6 cases, the programmatic specification somewhat captured the structure of the sentence, but were partially inaccurate, and in 2 cases, the resulting specifications were very incorrect.

Of the 62 sentences which did not have specifications, 6 were sentences which had unverifiable specifications. 12 sentences were not specifications which could reasonably be verified by this tool, and another 13 were somewhat beyond the abilities of our system to specify. 31 of these sentences could reasonably be specified, of which 14 described the initialization of an object, and 10 described side effects.

We review a few of the generated specifications below. In general, the programmatic specifications produced may not necessarily be useful programmatic specifications, but can still be useful as specifications which the user can review. Section 7.3 goes into detail about sentences which did not have generated specifications, including sentences with parse-trees.

## 7.2 Successful Specifications

```
/// Returns `(sin(x), cos(x))`
#[ensures(result == (sin(x), cos(x)))]
```

```
/// Computes `self + rhs`, returning `None` if overflow occurred.
#[ensures(overflows!(self + rhs) ==> (result == None))]
#[ensures(!overflows!(self + rhs) ==> (result == self + rhs))]
```

```
/// Returns `false` if `index` is greater than `self.len()`
#[ensures(index > self.len() ==> !result)]
```

We don't currently perform any steps to coerce types or identify the specific variables being referred to, which makes some of these specifications incomplete. However, assuming a correct implementation of those components, these examples demonstrate clear natural language specifications which can easily be turned into a programmatic specification with minimal modification required. Our system currently targets sentences in the "Return object", format, with possible "if" modifiers. It also supports disjoint returns in a single sentence, as in the second example. We also demonstrate that the system can detect "event occurs" syntax, allowing users to specify the behaviour of their system in terms of events triggered by a sequence of operations. The `overflows!` macro is a placeholder for a future implementation of a macro which can detect if an expression triggers an overflow.

### 7.2.1 Specifications Requiring Object Resolution

```
/// Returns the exponential of an `f32`
#[ensures(result == f32.exponential())]
```

```
/// Returns the natural logarithm of an `f32`
#[ensures(result == f32.natural_logarithm())]
```

These two specifications also demonstrate the lack of an object resolution system. In particular, such a system would use the query system to resolve the concrete functions, and determine that *`/// an `f32`.`* refers to a particular variable in the function signature. This also highlights a flaw in specifying functions which reference themselves indirectly in a circular fashion - in particular, both of these sentences are from functions that implement the underlying functionality for primitive methods. For example, the exponential documentation comes from `expf32`, which is equivalent to `f32::exp`.

### 7.2.2 Incorrect Specifications

```
/// The length of `src` must be the same as `self`
#[requires(src.length() == self.same())]
#[requires(src.length() == self)]
```

In this case, the system does not recognize that the length of `src` should be equal to the length of `self`, and instead detects that "must be the same" is equivalent to direct equality between the length of `src.length()` and `self`. This is a somewhat ambiguous specification, and it is plausible that a user might have either definition in mind.

```
/// Returns `None` if `self` and `other` are from different files
#[ensures((points_at!(other, different_file) && points_at!(self <
↪  different_file) ==> (result == None))]
```

The system can detect some cases of pointer style relationships between objects - however, it does not understand the concept of being from a file, which could be determine through object resolution, or the concept of "different" objects, which requires explicit support during the specification phase.

### 7.2.3 Identity or Trusted Specifications

```
/// Returns an iterator over the entries within a directory
#[ensures(result == directory.entry().iterator())]
```

This specification is largely flawed due to there only being one way to access the entries of a directory - through the function this documentation comes from. These "identity" specifications are not verifiable except by the user - in Prusti syntax, these should be considered trusted functions, annotated with `#[trusted]`, which Prusti will not verify and instead assume are correctly implemented. However, the system tries to create programmatic specifications for every sentence, and it does not account for the possibility of specifications which it can not verify. In these cases, the user could simply avoid annotating the function with `#[specify]`, and no specification will be generated. The function will still be available for use as a component of other specifications, even if it is not itself specified.

### 7.2.4 Unverifiable Specification

```
/// `transmute` should be the absolute last resort
#[requires(transmute == absolute_last_resort)]
```

This specification is an example of a specification which can not be verified automatically, as it requires the user to verify that the `transmute` operation is indeed the absolute last resort. However, this is still informative, even if it is not a valid programmatic specification.

### 7.2.5 Abstract Specification

```
/// Returns the path of a temporary directory
#[ensures(result == temporary_directory.path())]
```

This is an example of a specification which requires a more abstract understanding of the code, which is currently not possible with Prusti. In particular, it would require verifying that the path returned is indeed from a temporary directory, which requires a specification for what a temporary directory is. This in particular is information that would likely best be represented in the context of a temporary directory type. The system does not account for the possibility of these more abstract specifications.

## 7.3 Analyzing Sentences without Specifications

In this section, we inspect sentences which were not successfully transformed into specifications, and describe some of the reasons for why this might happen. We also consider some general statistics, to better understand the composition of sentences within documentation in typical programs.

We inspected 50 randomly selected sentences which generated no parse-tree. Of these, 2 sentences were malformed, and could not be specified. 3 were not specifications, and 1 was an ambiguous specification. 17 sentences were not verifiable specifications, or could not reasonably be verified by this tool. 10 were very difficult to specify, and would also require large amounts of code introspection. Three were somewhat specifiable, but would require features not yet in the scope of this project. 14 were specifications which a user could reasonably expect to be specified and verified. An example of each of these categories of specifications is provided below.

### 7.3.1 Malformed Sentence

```
/// Keywords which are usable in path segments (e
```

Programmatic specifications are generated from a single sentence at a time, and accordingly, documentation is split into individual sentences. This process does not currently handle cases where periods appear inside of brackets, and will split a sentence before it terminates, on the period within the brackets.

### 7.3.2 Ambiguous Specification

```
/// Float division that allows optimizations based on algebraic
↪  rules.
```

This sentence is a specification, which is partially verifiable, but also ambiguous in how a programmatic specification should be generated. In particular, partial matching of the string should be able to determine that "Float division" specifies the behaviour of the function, even if it is not necessarily possible to verify or specify "allows optimizations based on algebraic rules" in Prusti.

### 7.3.3 Challenging Specifiable Sentence

```
/// Like `wait`, the lock specified will be re-acquired when this
↪  function returns, regardless of whether the timeout elapsed
↪  or not.
```

This sentence specifies the behaviour of its function, but is well beyond the capabilities of this system to specify. Additionally, there is no way to specify time-dependent statements within Prusti, which currently makes this sentence unverifiable.

### 7.3.4 Non-specification Sentence

```
/// For more information on the meaning and layout of the
↪  `flowinfo` and `scope_id` parameters, see IETF RFC 2553,
↪  Section 3.
```

This sentence is not a specification, and instead falls into the category of sentences which serve as general documentation for the underlying function.

### 7.3.5 Specifiable, Verifiable Sentence

```
/// Create a new `Context` from a `&Waker`.
```

```
/// Sets the position of this cursor.
```

The system currently does not support most specifications for operations with side effects, including the initialization of items and setting values, as seen above. In some cases, it can successfully generate parse-trees for such sentences, but these parse-trees can not be compiled into programmatic specifications. However, a user could reasonably expect that sentences in this style could be verified.

# 8 Discussion and Future Work

In conclusion, this project is a first effort towards automatic conversion of natural language specifications into programmatic specifications for the documentation case. Such a system would greatly increase the accessibility of program verification tools, helping users develop more complex code bases in a more

robust fashion by reducing the occurrence of incorrectly implemented and incorrectly documented software. Some next steps are described below, some of which are intended to address limitations in the current system.

The two most immediate future changes we would like to make are implementing core portions of this project as an extension to the Rust compiler, and utilizing the query system when generating specifications. Rust compiler extensions have access to Rust's type-inference tools, macro expansion features, and can perform more accurate discovery of available types when specifying functions. Type-inference would allow determining the type of a code block in Rust documentation, which is useful for both the CFG, which could incorporate this type information to create more specific parse-trees, and for the query system, which could use the type information to return more specific results. Additionally, integrating the query system into the specification phase would allow more function and struct references to be automatically resolved, giving users the flexibility to freely use natural language when specifying their code.

One major limitation of this project is the data which was considered in designing the CFG. Our CFG does not cover a large number of cases, which prevents it from matching particular styles of specifications, and it does not incorporate any statistical information about the relative frequency at which productions occur, which prevents usage of probabilistic parsing methods to generate the likeliest parse-tree for a particular sentence. Expanding the grammar can be accomplished with more thorough analysis of documentation from both the Rust standard library and a number of the most popular libraries, or crates, in the Rust ecosystem. Afterwords, frequencies can be generated for the grammar components using statistical analysis of the parse-trees resulting from applying the grammar to the aforementioned sources.

Another potential way to improve the CFG used in this project is to integrate the output of NER and SRL analysis directly into a CFG. The NER and SRL models we use in this project can detect entities and predicates in a flexible fashion, but do not have any mechanism to determine the overall structure of the text, and do not detect important syntactic elements of specifications, such as if else blocks. In contrast, the CFG is not flexible when detecting entities and predicates, but is very effective at detecting and representing the syntactic elements of specifications. By combining the two systems, this project could generalize to a much larger number of natural language specifications, without a corresponding increase in the complexity of the CFG.

We would also like to consider training models for POS tagging, NER, and SRL, on domain specific data - in this case, Rust documentation, or using models trained with more data. In particular, [Tabassum et al. 2020] have created word vectors and an NER model, trained on a large corpus of StackOverflow questions. Comparing the performance of this model against the NER model we use, which frequently produces incomplete annotations, would help illuminate whether a lack of training data limits the models we use in this paper, or whether it is necessary to train a model on more specific data. Using domain specific word vectors could also improve the performance of the query system, which currently uses WordNet's synonym sets to determine whether two words are equivalent.

This means that users may not find relevant functions for a particular query string. For example, WordNet does not consider "remove" and "delete" to be synonyms, even though these can be frequently interchanged. However, the domain specific word vectors for these words and other such cases should reflect this similarity, which would hopefully allow the query system to return relevant results in more cases.

In a future work, it would also be useful to determine if a significant number of natural language specifications contain discontinuous entities (Fig 17 provides some examples of this for the "shift right" operation). Most off-the-shelf solutions for NER do not currently support discontinuous entities, including the one we use in this project, which may need to be accounted for.

1. `/// Shift `a` to the right by `b`.`
2. `/// `a` is shifted to the right by `b`.`
3. `/// `a` is right shifted by `b`.`

Figure 17: Invocations of the shift right function, or >> operation.

Our system also currently assumes that any sentence that can be parsed into a tree is a programmatic specification. However, this assumption is not necessarily true, as seen in Section 7.2.4. Accordingly, it would be useful to be able to determine if a specification is programmatic or not, as this would prevent false-positive programmatic specifications from being generated and inserted into the annotated output file. In addition, the non-programmatic specifications could be relevant for the user, who could manually verify that their implementation, or usage of a particular function is indeed correct.

This system could also be extended beyond simply creating programmatic specifications from documentation, and provide feedback about program documentation as a whole. For example, it is possible for users to create specifications for their functions such that they do not specify interesting aspects of the function's execution, or are always true, such as "`true` is equal to `true`". Providing a feature that can measure the quality of a particular specification, using factors such as whether it is always true or ambiguous, could help users build better documentation, and verify that the specifications they provide are actually useful. This could further be integrated into the specification pass, allowing users to seamlessly update their documentation to be more clear, and increasing the overall percentage of natural language specifications converted into programmatic specifications.

# A Part-of-Speech Tags

## A.1 OntoNotes 5 / Penn Treebank

Table 1: Part-of-Speech Tags (OntoNotes 5 / Penn Treebank)

| POS Tag | Description |
|---|---|
| . | punctuation mark, sentence closer |
| , | punctuation mark, comma |
| -LRB- | left round bracket |
| -RRB- | right round bracket |
| “ | opening quotation mark |
| ”” | closing quotation mark |
| ″ | closing quotation mark |
| : | punctuation mark, colon or ellipsis |
| $ | symbol, currency |
| # | symbol, number sign |
| AFX | affix |
| CC | conjunction, coordinating |
| CD | cardinal number |
| DT | determiner |
| EX | existential there |
| FW | foreign word |
| HYPH | punctuation mark, hyphen |
| IN | conjunction, subordinating or preposition |
| JJ | adjective (English) |
| JJR | adjective, comparative |
| JJS | adjective, superlative |
| LS | list item marker |
| MD | verb, modal auxiliary |
| NIL | missing tag |
| NN | noun, singular or mass |
| NNP | noun, proper singular |
| NNPS | noun, proper plural |
| NNS | noun, plural |
| PDT | predeterminer |
| POS | possessive ending |
| PRP | pronoun, personal |
| PRP$ | pronoun, possessive |
| RB | adverb |
| RBR | adverb, comparative |
| RBS | adverb, superlative |
| RP | adverb, particle |
| TO | infinitival “to” |
| UH | interjection |

Table 2: Part-of-Speech Tags (OntoNotes 5 / Penn Treebank) - continued

| POS Tag | Description |
|---|---|
| VB | verb, base form |
| VBD | verb, past tense |
| VBG | verb, gerund or present participle |
| VBN | verb, past participle |
| VBP | verb, non-3rd person singular present |
| VBZ | verb, 3rd person singular present |
| WDT | wh-determiner |
| WP | wh-pronoun, personal |
| WP$ | wh-pronoun, possessive |
| WRB | wh-adverb |
| SP | space (English) |
| ADD | email |
| NFP | superfluous punctuation |
| GW | additional word in multi-word expression |
| XX | unknown |
| BES | auxiliary “be” |
| HVS | forms of “have” |
| _SP | whitespace |

These POS tags and definitions are sourced from “spaCy” [Honnibal et al. 2020], which in turn sources them from the Penn Treebank [Taylor, Marcus, and Santorini 2003].

### A.2 Program Specification POS tags

Table 3: Part-of-Speech Tags (Program Specifications)

| POS Tag | Description |
|---|---|
| CODE | block of code, surrounded with `symbols |
| LIT | Rust literal, any type |
| RET | word "return" |
| IF | word "if" |
| FOR | word "for" |
| ARITH | arithmetic operation |
| SHIFT | left or right shift |
| DOT | . symbol (or . POS tag) |
| COMMA | , symbol (or , POS tag) |
| EXCL | ! symbol |

These POS tags and definitions are applied during the retokenization procedure in Section 5.3. CODE and LIT are custom tokens which have no equivalent in spaCy's POS tags. RET, IF, FOR, ARITH, and SHIFT are more specific POS tags applied to improve the specificity of the grammar. The DOT, COMMA, and EXCL tags are specifically applied to ensure compatibility with the file format used by nltk's ChartParser.

## B Context-free Grammar

Listing 1: The context-free grammar used in this project. Each line is annotated with a description of the production rule. Certain production rules use regex-style syntax for brevity.

```
# Specification root
S → MRET | RETIF | HASSERT | QASSERT | SIDE | FNCALL

## Wrapping default POS tokens for more general parsing
# Noun with arbitrarily many adjectives
MNN → (JJ)* NN(S|P|PS)?
# Adjective with optional determinant, such as "the"
TJJ → (DT)? JJ(R|S)?
# Adjective with optional modifier
MJJ → (RB)? JJ(R|S)?
# Verb with optional modifier
MVB → (RB)? VB(Z|P|N|G|D)?
# If and only if
IFF → IF CC RB IF
# than or equal to (eg. greater EQTO)
```

```
EQTO → IN CC JJ IN

## Operations
# Modified boolean operation (bitwise/logical and / or)
BITOP → JJ CC | NN CC
# Arithmetic operation
ARITHOP → ARITH IN | ARITH
# Shift operation
SHIFTOP → SHIFT IN DT NN IN | JJ SHIFT
# Any operation
OP → BITOP | ARITHOP | SHIFTOP

# Objects
OBJ → PRP | (DT)? MNN | CODE | (DT)? LIT
| OBJ OP OBJ | (DT)? FNCALL | DT VBG MNN | PROP_OF OBJ

## Object relationships
# Relationship to object
# less than object, less than or equal to object,
# is object, is less than a or b
REL → TJJ IN OBJ | TJJ EQTO OBJ | IN OBJ | REL CC OBJ
# Modified relationship
# not less than object
MREL → RB REL | REL
# is adjective, is relation, is a literal (eg. ==)
PROP → MVB MJJ | MVB MREL | MVB OBJ | MVB | MVB RANGEMOD
# A property of an item
# A value of
PROP_OF → DT MNN IN DT | DT MNN IN | DT MJJ IN | DT MJJ IN DT

## Ranges
# Range separator, between range start and range end
# between a AND b, from a TO b
RSEP → CC | IN | TO
# Range
# OBJECT from START to END, from START to END, up to END
RANGE → OBJ IN OBJ RSEP OBJ | IN OBJ RSEP OBJ | IN IN OBJ
# Range with optional modifier on range end
# eg. from a to b INCLUSIVE
RANGEMOD → RANGE | RANGE COMMA JJ | RANGE JJ

## Assertions of fact
# Object is something, object 1 and object 2 are something
ASSERT → OBJ PROP | OBJ CC ASSERT
# Object must be something, object 1 and object 2 must be something
HASSERT → OBJ MD PROP | OBJ CC HASSERT
```

```
# For some object
QUANT → FOR DT OBJ
# Quantifier, with optional range and modifiers
# For all objects (in range)? (with some property)?,
For all objects in range and with some property
QUANT_EXPR → QUANT | QUANT RANGEMOD | QUANT_EXPR COMMA CC MREL
| QUANT_EXPR COMMA MREL | QUANT_EXPR CC MREL | QUANT_EXPR MREL
# Quantifier, with assertion over items
# For all items in range, some expression must be true
QASSERT → QUANT_EXPR COMMA HASSERT | QUANT_EXPR HASSERT
| HASSERT QUANT_EXPR | CODE QUANT_EXPR

## Return operations
# Return object
MRET → RET OBJ | OBJ VBZ RET | OBJ RET
# Items which represent boolean condition
BOOL_EXPR → ASSERT | QASSERT | CODE | FNCALL
# If or IFF boolean is true
COND → IF BOOL_EXPR | IFF BOOL_EXPR
# Return obj if condition is satisfied
# Return object if condition
RETIF → MRET COND | COND COMMA MRET
| RETIF COMMA RB RETIF | RETIF COMMA RB OBJ
| MRET COMMA MRET COND

## Side effects
SIDE → OBJ VBZ MVB IN OBJ | OBJ VBZ MVB
| OBJ VBZ MJJ MVB IN OBJ | VBZ TJJ OBJ IN OBJ CC MRET
# Object verbed – eg. `a` is negated, `a`
OBJV → OBJ MVB | OBJ VBZ MVB
# Verbs object – eg. negates `a`
VOBJ → VBZ OBJ IN OBJ | VBZ OBJ TO OBJ
| VBZ OBJ | VBZ DT OBJ | VBZ DT JJ OBJ
# Noun/event occurred – eg. "overflow occured"
EVENT → MNN VBD
```

# References


Astrauskas, V. et al. (2019). "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. ACM, 147:1–147:30. DOI: 10.1145/3360573.

Cambria, Erik and Bebo White (2014). "Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]". In: *IEEE Computational Intelligence Magazine* 9.2, pp. 48–57. DOI: 10.1109/MCI.2014.2307227.

Filliâtre, Jean-Christophe (2011). *Deductive software verification*.

Ghosh, Shalini et al. (2016). *ARSENAL: Automatic Requirements Specification Extraction from Natural Language*. arXiv: 1403.3142 [cs.CL].

Gouy, Isaac. *Which programming language is fastest?* URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

Honnibal, Matthew et al. (2020). *spaCy: Industrial-strength Natural Language Processing in Python*. DOI: 10.5281/zenodo.1212303. URL: https://doi.org/10.5281/zenodo.1212303.

Loper, Edward and Steven Bird (2002). "NLTK: The Natural Language Toolkit". In: *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008). *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press. ISBN: 978-0-521-86571-5. URL: http://nlp.stanford.edu/IR-book/information-retrieval-book.html.

Manning, Christopher D. et al. (2014). "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

Mitkov, Ruslan (2005). *The Oxford Handbook of Computational Linguistics (Oxford Handbooks)*. USA: Oxford University Press, Inc. ISBN: 019927634X.

Müller, P., M. Schwerhoff, and A. J. Summers (2016). "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, pp. 41–62.

Màrquez, Lluís et al. (June 2008). "Semantic Role Labeling: An Introduction to the Special Issue". In: *Computational Linguistics* 34.2, pp. 145–159. ISSN: 0891-2017. DOI: 10.1162/coli.2008.34.2.145. eprint: https://direct.mit.edu/coli/article-pdf/34/2/145/1798596/coli.2008.34.2.145.pdf. URL: https://doi.org/10.1162/coli.2008.34.2.145.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

*Primitive type i32*. URL: https://doc.rust-lang.org/std/primitive.i32.html (visited on 07/30/2021).

*Primitive type u32*. URL: https://doc.rust-lang.org/std/primitive.u32.html (visited on 08/31/2021).

Puccetti, Armand, Gaël de Chalendar, and Pierre-Yves Gibello (2021). "Combining Formal and Machine Learning Techniques for the Generation of JML Specifications". In: Association for Computing Machinery. ISBN: 9781450385435.

Pullum, Geoffrey K. and Gerald Gazdar (1987). "Natural Languages and Context-Free Languages". In: *The Formal Complexity of Natural Language*. Ed. by Walter J. Savitch et al. Dordrecht: Springer Netherlands, pp. 138–182. ISBN: 978-94-009-3401-6. DOI: 10.1007/978-94-009-3401-6_7. URL: https://doi.org/10.1007/978-94-009-3401-6_7.

Tabassum, Jeniya et al. (July 2020). "Code and Named Entity Recognition in StackOverflow". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, pp. 4913–4926. DOI: 10.18653/v1/2020.acl-main.443. URL: https://aclanthology.org/2020.acl-main.443.

Taylor, Ann, Mitchell Marcus, and Beatrice Santorini (2003). *The Penn Treebank: An Overview*.