University Politehnica of Bucharest

Faculty of Automatic Control and Computer Science

Pattern-Based Syntax Analysis for Julia

Author: Iulia Maria Dumitru

Coordinator: Șl. Dr. Ing. Mihnea Muraru

Department of Computer Science and Engineering

June 2025

Abstract

In this thesis we present the implementation of a system for matching syntax and writing static analysis rules in Julia. We begin by describing Julia's metaprogramming capabilities and comparing them to syntax manipulation libraries from the Racket ecosystem. We explain the rationale for porting part of Racket's functionality to Julia. After understanding the main principles behind the inspiration libraries, we present our Julia implementation. The defined concepts and algorithms are illustrated with instructive examples and the system's flexibility is showcased in a real-world scenario. Finally, we outline potential extensions and improvements for our implementation.

Rezumat

În această lucrare prezentăm implementarea unui sistem de analiză sintactică pentru limbajul de programare Julia. Începem prin a descrie capabilitățile de metaprogramare ale limbajului Julia, comparându-le cu două biblioteci de manipulare de sintaxă din ecosistemul limbajului Racket. Motivăm decizia de a implementa o parte din funcționalitatea acestor biblioteci în Julia. După ce prezentăm principiile care stau la baza bibliotecilor din Racket, propunem o implementare a acestora în Julia. Ilustrăm cu exemple clarificatoare conceptele și algoritmii pe care îi definim. Pentru a demonstra flexibilitatea sistemului conceput, îl folosim pentru a rezolva o problemă practică. În final, expunem potențiale extensii și îmbunătățiri ale implementării noastre.

Acknowledgements

I am grateful to JuliaHub for providing me with the necessary context for carrying out this endeavour. I would also like to thank my partner, Andrei, for the invaluable support he gave me throughout the entire process. His technical advice and aesthetic recommendations not only improved the quality of this work, but also strengthened my software design skills. My gratitude goes to my coordinating professor as well, whose courses have inspired me to pursue the challenging yet rewarding field of compilers. Last but not least, I want to thank my colleagues and friends who have read iterations of this paper in great detail and pointed out the mistakes and ambiguities they encountered.

Contents

1 Introduction

The ability to reason about *code as data* is essential for writing static analysis tools and is the quintessence of metaprogramming. Compilers, interpreters, linters — they all manipulate program inputs, whether it's for generating other programs, evaluating expressions or signaling errors.

Languages that facilitate access to their own representation are generally highly extendable and introspectable. Lisp is the canonical example of a *homoiconic* language, with S-expressions having been invented specifically for unifying the representation of Lisp code and data [1]. Lisp dialects rely on metaprogramming (macros) to analyse, generate and transform programs dynamically.

Julia takes inspiration from Lisp in its "willingness to represent itself" [2]. Like Lisp, it has a powerful macro system that enables manipulating programs at the syntactic level [3]. But unlike Lisp, Julia is a young language [4] and has much to learn from its older relatives in order to fully take advantage of its own features.

1.1 Syntax Manipulation in Julia

Julia programs are represented as sequences of expressions — objects of type `Expr`. Expressions have a head and a list of arguments, which are themselves expressions[1]. The Julia parser — JuliaSyntax.jl[2] — parses text and transforms it into syntax trees. These can be directly `Expr`s or other tree representations such as `SyntaxNode`s. The parser does not directly produce syntax trees, allowing its output to be post-processed into any custom tree data structure [5].

`SyntaxNode` data structures are richer than `Expr`s. They provide detailed code provenance information and remain more faithful to the code's textual representation [6]. Code analysis tools such as JET.jl [7] or Cthulhu.jl [8] use and extend `SyntaxNode`s for more flexibility and expressivity.

There are already a few syntax manipulation packages in the Julia ecosystem. Some are widely used, including MacroTools.jl [9] and MLStyle.jl [10]. However, they don't fully exploit Julia's representational flexibility. Inspiration can be drawn from research-oriented Lisp dialects, which explore syntax matching and templating more extensively.

[1] Symbols and atoms (`QuoteNode`s) are also representations of Julia code. For conciseness, we refer to these as expressions as well.

[2] JuliaSyntax has been the default Julia parser since Julia 1.10 (2023). Before JuliaSyntax, the default parser was implemented in femtolisp.

1.2 Syntax Manipulation in Racket

Racket stands out among Lisps for being "a programming language for creating new programming languages" [11]. It provides a powerful mechanism for parsing syntax and creating new syntactic forms within Racket itself through the `syntax/parse` library [12].

The main focus of `syntax/parse` is writing robust macros that automatically generate specific error messages for invalid input. Macros are syntactic specifications defined with *syntax patterns* and *syntax templates*. Patterns match pieces of syntax, which get transformed into other pieces of syntax at compile time. The transformation consists of filling in templates with *pattern variables* bound during pattern matching. Patterns can reference *syntax classes* for constraining pattern variables. Syntax classes can be seen as abstractions over patterns that encapsulate syntactic categories [13].

This framework forms the basis for Resyntax [14], a Racket refactoring library that allows writing and using suites of rules for analysing and rewriting code. It provides a command line tool with two commands: `resyntax analyze` and `resyntax fix`. The former analyses files and prints suggestions specified in the refactoring rules driving the analysis. The latter applies the suggestions directly to the analysed code and prints a summary of the refactoring.

1.3 Bringing Racket's Philosophy to Julia

There is no technical limitation stopping Julia from attaining the same level of expressivity as Racket when it comes to syntax specification and analysis. Julia's code representation can be reflected upon and restructured. Julia's macro system can be enriched with pattern-matching macros. And static analysis tools for Julia can build upon these macros to create versatile and reliable frameworks for enforcing code standards.

This paper presents Argus.jl, a Julia package for pattern-based static analysis inspired by Racket's `syntax/parse` and Resyntax libraries.

2 Understanding the Inspiration Model

2.1 Background and Motivation Behind `syntax/parse`

Before `syntax/parse`, the state of the art for writing macros was Macro-by-Example (MBE) [15]. MBE was developed as a Lisp specification language allowing a declarative specification of macros[3]. In their 1986 paper introducing the notation, Kohlbecker and Wand offer an example with the familiar `let` expression implemented as a lambda function application[4]:

```
(let ((i e) ...) b ...)
  ==> ((lambda (i ...) b ...) e ...)
```

This specification states that a `let` expression expects zero or more *binding pairs* (`(i e) ...`) and zero or more body expressions (`b ...`). It is semantically equivalent to a lambda function with zero or more formal parameters (`i ...`) — corresponding to the bound variables in the `let` form — and a body containing zero or more expressions (`b ...`), applied to zero or more actual parameters (`e ...`). Defining `let` as a macro without the MBE notation requires carefully destructuring and reassembling input code:

```
(defmacro let (decls . body)
  `((lambda ,(map car decls) . ,body)
    . ,(map cadr decls)))
```

It is difficult to distinguish the macro's input and output shapes by reading this definition. Validating the input in order to generate useful error messages would further complicate the definition, making it almost incomprehensible[5].

With MBE, a `let` definition can resemble the specification much more closely:

```
(declare-syntax let
  [(let ((i e) ...) b ...)
   ((lambda (i ...) b ...) e ...)])
```

The syntactic shape of the input is validated automatically through pattern matching. This eliminates the need to write explicit code for structural validation. The variables bound when matching the *pattern* — `(let ((i e) ...) b ...)` — are used to fill in the *template* — `((lambda (i ...) b ...) e ...)` — and generate the resulting code. Additional checks are still necessary to ensure the correct use of a `let` expression.

[3] The concept gained popularity and started being implemented in other languages, most famously in Rust.

[4] All examples in this section are either adaptations or direct reproductions from the cited sources.

[5] The definition is already incomprehensible for the reader who is not familiar with Lisp.

For example, there should be a condition verifying that all bound identifiers are unique. MBE lacks a straightforward representation for such conditions.

In the 2012 article "Fortifying Macros" [16], Ryan Culpepper explains in more detail the limitations of the MBE notation, presenting a strong case for creating a new system. A `let` macro that fully validates its input can be written clearly and concisely in this new framework:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let ([var:identifier rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate #'(var ...))
                 "duplicate variable name"
     #'((lambda (var ...) body) rhs ...)]))
```

First, `syntax-parse` verifies that the input syntax matches the pattern in the first clause of its body — `(let ([var:identifier rhs:expr] ...) body:expr)`. Then, it asserts that there are no duplicates in the identifier list bound to `var`. If there are, the parsing fails with the message "duplicate variable name". Finally, if all checks succeeded the input is transformed into the new syntax specified by the template in the last clause — `#'((lambda (var ...) body) rhs ...)`.

Given correct input, the above `let` implementation expands to the corresponding lambda application at compile time and evaluates the application at run time. Given incorrect input, a specific error is thrown at compile time.

```
(let ([a 1] [b 2]) (+ a b))
  compile-time expansion: ((lambda (a b) (+ a b)) 1 2)
  run-time result: 3
(let (["a" 1]) (add1 a))
  compile-time error: expected identifier
(let ([a 1] [a 2]) (+ a a))
  compile-time error: duplicate variable name
```

Misuses that are not explicitly covered by the `syntax-parse` definition generate a generic error message.

```
(let (x 5) (add1 x))
  compile-time error: bad syntax
```

2.2 Building Blocks

The layout of a macro definition in `syntax/parse` consists of *patterns* and *templates*. Patterns match input code, binding *pattern variables* in the process. Pattern variables can be annotated by *syntax classes* in order to refine matching and error-reporting.

The definition of `let` in the previous sub-section contains three pattern variables: `var`, `rhs` and `body`. Because `var` and `rhs` are surrounded by an *ellipsis*, they have an *ellipsis depth* of 1. `body` does not have any ellipses around it and has ellipsis depth 0. Ellipses can be nested to any depth. A variable appearing in the pattern at ellipsis depth d must appear in the template at the same depth.

All pattern variables in the `let` pattern have explicit syntax class annotations — `expr` for `rhs` and `body` and `identifier` for `var`. A syntax class groups together "syntactically similar" forms and associates them with a corresponding class of errors.

Error reporting for `let` can be further improved by defining a syntax class corresponding to binding pairs and rewriting `let`:

```
(define-syntax-class binding
  #:description "binding pair"
  (pattern [var:identifier rhs:expr]))

(define-syntax (let stx)
  (syntax-parse stx
    [(let (b:binding ...) body:expr)
     #:fail-when (check-duplicate #'(b.var ...))
                 "duplicate variable name"
     #'((lambda (b.var ...) body) b.rhs ...)]))
```

If `b` binds to anything other than a binding pair, a specific error message is shown rather than the generic "bad syntax":

```
(let (x 5) (add1 x))
  compile-time error: expected binding pair
```

2.3 Resyntax

Even though `syntax/parse` is centered around writing macros, it can also form the basis for tools that identify problematic patterns in code. The Resyntax library [14] leverages its capabilities for matching patterns and rewriting code to implement such a tool.

Resyntax allows the definition of *refactoring rules*, which are "macro-like functions defined in terms of `syntax-parse` that specify how to search for and refactor different coding patterns". A rule matches patterns in source code and makes a refactoring suggestion with an associated description.

The following rule searches for unnecessarily nested `or` expressions:

```
(define-refactoring-rule nested-or-to-flat-or
  #:description "This nested ‘or‘ expression can be flattened."
  #:literals (or)
  (or a (or b c))
  (or a b c))
```

The first line in the body contains the rule's description. The second line provides a list of identifiers that should be treated as literals instead of pattern variables. The third and fourth lines define the pattern to be searched for and the template for generating the refactored `or` expression, respectively.

Refactoring rules can be grouped in *refactoring suites*. Resyntax comes with a default refactoring suite. The user is free to use the default or define custom rules and suites to fit their needs.

3 Adapting the Model to Julia

Argus.jl was created to explore Julia's ability to offer highly expressive syntax manipulation tools similar to Racket's. It extends the Julia parser with a custom tree data structure for representing patterns. Its pattern matching capabilities exceed the scope of existing pattern matching Julia packages such as MacroTools.jl [9]. Like Resyntax, Argus builds upon the syntax matching framework to create a powerful mechanism for writing static analysis rules.

The scope of Argus is currently limited to matching syntax and writing rules that search for patterns in code. However, it can be easily extended to include templating facilities for code refactoring.

3.1 Overview of Julia's Syntax

The effectiveness of a metaprogramming tool is strongly influenced by the extent of the information it can access. Julia's surface syntax `Expr`s are annotated with `LineNumberNodes` to provide source location information [17]. This information is limited to line number and file name. The code structure can be investigated by looking at the head and arguments of its `Expr` representation.

The following examples illustrate the structure of `Expr`s. The expressions are evaluated in the Julia REPL [18].

```
julia> toplevel_expr = Meta.parseall("""
                                     x = :x
                                     println(x, 2)
                                     """)
:($(Expr(:toplevel, :(#= none:1 =#), :(x = :x), :(#= none:2 =#), :(println(x, 2)))))

julia> toplevel_expr.head
:toplevel

julia> toplevel_expr.args
4-element Vector{Any}:
 :(#= none:1 =#)
 :(x = :x)
 :(#= none:2 =#)
 :(println(x, 2))
```

`Meta.parseall` parses a series of toplevel expressions into an `Expr` with a `:toplevel` head[6].

[6] The equivalent for single expressions is `Meta.parse`.

`LineNumberNodes` are printed inside block comments (`#=...=#`). They indicate the file in which the annotated expression was executed (`none` indicates REPL execution) and the line at which it appears in the source text.

`x = :x` is an assignment expression with two children — the identifier `x` and the `Symbol :x`. `println(x, 2)` is a function call with three children — the function name, `println`, and two function arguments, `x` and `2`.

```
julia> toplevel_expr.args[2].head
:(=)

julia> toplevel_expr.args[2].args
2-element Vector{Any}:
 :x
 :(:x)

julia> toplevel_expr.args[4].head
:call

julia> toplevel_expr.args[4].args
3-element Vector{Any}:
  :println
  :x
2
```

Extracting information from `LineNumberNode`s can be unintuitive and limiting. A `LineNumberNode` is not directly attached to the expression it annotates. Instead, it appears as a sibling node located immediately above the corresponding expression. Moreover, it has no knowledge of the expression's textual representation.

`Expr`s can also cause confusion. In an `Expr` an identifier is represented as a `Symbol`. A `Symbol` is a `QuoteNode`. Literals are anything other than `Expr`, `Symbol` and `QuoteNode`.

```
julia> typeof(:( x = 2 ))
Expr

julia> typeof(:( x ))
Symbol

julia> typeof(:( :x ))
QuoteNode

julia> typeof(:( "abc" ))
String
```

JuliaSyntax's `SyntaxNode`s facilitate access to code-related information. They retain all syntactic trivia such as whitespace and comments [19], making it possible to determine an expression's byte span. The type of a `SyntaxNode` is denoted as a "syntax kind" [20], allowing the efficient definition of predicates such as `is_identifier` and `is_operator`.

```
julia> using JuliaSyntax: JuliaSyntax, SyntaxNode, parsestmt, parseall, kind

julia> toplevel_syntax_node = parseall(SyntaxNode, """
                                                   x = :x
                                                   println(x, 2)
                                                   """)
SyntaxNode:
[toplevel]
  [=]
    x                                   :: Identifier
    [quote-:]
      x                                 :: Identifier
  [call]
    println                             :: Identifier
    x                                   :: Identifier
    2                                   :: Integer


julia> kind(toplevel_syntax_node)
K"toplevel"

julia> toplevel_syntax_node.children
2-element Vector{SyntaxNode}:
 (= x (quote-: x))
 (call println x 2)

julia> kind(toplevel_syntax_node.children[1])
K"="

julia> kind(toplevel_syntax_node.children[2])
K"call"

julia> JuliaSyntax.source_location(toplevel_syntax_node.children[2])
(2, 1)

julia> JuliaSyntax.byte_range(toplevel_syntax_node.children[2])
8:20

julia> JuliaSyntax.is_identifier(parsestmt(SyntaxNode, "x"))
true

julia> JuliaSyntax.is_identifier(parsestmt(SyntaxNode, ":x"))
false

julia> JuliaSyntax.is_literal(parsestmt(SyntaxNode, "2"))
true
```

The parsing functions from JuliaSyntax can produce both `Expr`s and `SyntaxNode`s, among other tree types [19]. The result type is specified by the first argument.

A `SyntaxNode` is implemented as a concrete subtype of the generic type `TreeNode`. `TreeNode`s can be extended to create custom syntactic representations.

```
mutable struct TreeNode{NodeData}
    parent::Union{Nothing, TreeNode{NodeData}}
    children::Union{Nothing, Vector{TreeNode{NodeData}}}
    data::Union{Nothing, NodeData}

    ...
end

const SyntaxNode = TreeNode{SyntaxData}
```

`SyntaxData` is a data structure that holds information such as source location and syntax kind. Its implementation is not important for our purposes.

Because of the advantages of `SyntaxNode` over `Expr`, Argus uses `SyntaxNode`s to represent Julia code.

3.2 Implementing Argus.jl

Argus can be split in two distinct components:

- a syntax matching mechanism;
- a framework for writing rules.

The former provides the necessary building blocks for defining an *embedded domain-specific language* [21] for syntax matching. It implements the fundamental concepts behind `syntax/parse`, borrowing syntactic representation for some of the constructs.

The latter extends this syntax matching language with implementations for *rules* and *rule groups*. These are inspired by Resyntax's *refactoring rules* and *refactoring suites* but are currently limited to searching for patterns in code.

3.2.1 Matching Syntax

A syntax matching engine needs to define two essential concepts: *patterns* and the *criteria for compatibility* between a pattern and a syntax object.

Argus defines patterns in terms of regular abstract syntax tree nodes (hereafter referred to as simply *syntax nodes*) and special nodes denoting *pattern forms* [22]. Pattern nodes are implemented by extending the `TreeNode` type described in section 3.1 with a `SyntaxPatternNode` type. A `SyntaxPatternNode` can either carry regular `SyntaxNode` data or pattern form-specific data.

```
const SyntaxPatternNode =
    JuliaSyntax.TreeNode{Union{JuliaSyntax.SyntaxData, AbstractPatternFormSyntaxData}}
```

A `Pattern` is a light wrapper around a `SyntaxPatternNode`.

```
struct Pattern
    src::SyntaxPatternNode
end
```

`Pattern`s and `SyntaxPatternNode`s only differ in their level of abstraction. A `Pattern` represents the higher level notion of a *syntax pattern*. A `SyntaxPatternNode` is the implementation of a syntax pattern — its representation as an *abstract syntax tree*. Since we are discussing implementation details, we will be using the terms *pattern* and *pattern node* interchangeably.

Pattern Forms

Argus implements five pattern forms. When defining patterns, a pattern form can appear explicitly as `~<form-name>(<form-args>...)` or implicitly in its syntactic-sugar notation.

`~var`

```
struct VarSyntaxData <: AbstractPatternFormSyntaxData
    var_name::Symbol
    syntax_class_name::Symbol
end
```

A `~var` form associates a pattern variable with a syntax class. The following representations of `~var` are equivalent:

```
julia> @pattern ~var(:x, :expr)
Pattern:
x:::expr                                :: ~var

julia> @pattern {x:::expr}
Pattern:
x:::expr                                :: ~var

julia> @pattern {x}
Pattern:
x:::expr                                :: ~var
```

Syntax classes referenced by a `~var` pattern need to be registered in a *syntax class registry* before match time[7].

[7]The notion of a syntax class registry is general and its implementation is not important. For the curious reader, Argus represents it as a dictionary mapping syntax class names to definitions.

~fail

```
struct FailSyntaxData <: AbstractPatternFormSyntaxData
    condition::Function
    message::String

    FailSyntaxData(cond, msg::String) = new(fail_condition(cond), msg)
end
```

A `~fail` form causes a pattern match to fail if a given condition is satisfied, providing a relevant failure message. The condition can reference previously-bound pattern variables.

```
julia> fail = @pattern ~fail(true, "always fails")
Pattern:
[~fail]
  true                                  :: Bool
  "always fails"                        :: String

julia> syntax_match(fail, parsestmt(SyntaxNode, "dummy"))
MatchFail("always fails")

julia> fail_if_even = @pattern ~and(
           {n},
           ~fail(iseven(n.value), "expected odd")
       )
Pattern:
[~and]
  n:::expr                              :: ~var
  [~fail]
    [call]
      iseven                            :: Identifier
      [.]
        n                               :: Identifier
        value                           :: Identifier
    "expected odd"                      :: String

julia> syntax_match(fail_if_even, parsestmt(SyntaxNode, "2"))
MatchFail("expected odd")

julia> syntax_match(fail_if_even, parsestmt(SyntaxNode, "3"))
BindingSet with 1 entry:
  :n => Binding:
          Name: :n
          Bound source: 3 @ 1:1
          Ellipsis depth: 0
          Sub-bindings:
            BindingSet with 0 entries
```

The `FailSyntaxData` constructor expects a condition expression and a failure message. The condition expression is processed into an anonymous function that takes as argument a set of pattern variable bindings. During matching, when the function is called, the

pattern variables are defined in its evaluation context. The condition expression is then evaluated in the same context. All pattern variables referenced in the fail condition should have associated bindings at match time.

`~fail` can be written as a `@fail` macro inside a pattern. Pattern fail conditions in the `@fail` form require the pattern to be written as a sequence of expressions using a `begin ... end` block [23]. The expressions above the first fail condition are interpreted as toplevel expressions. Fail conditions and toplevel expressions are combined into an `~and` pattern. They cannot be interspersed.

```
julia> @pattern begin
           expr1
           expr2
           @fail condition "message"
       end
Pattern:
[~and]
  [toplevel]
    expr1                                :: Identifier
    expr2                                :: Identifier
  [~fail]
    condition                            :: Identifier
    "message"                            :: String

julia> @pattern begin
           expr1
           @fail condition "message"
           expr2
       end
ERROR: LoadError: SyntaxError: invalid ‘@pattern‘ syntax
Pattern expressions and fail conditions cannot be interspersed.
...
```

The `@fail` macro has no meaning outside of pattern definitions.

```
julia> @fail condition "message"
ERROR: LoadError: UndefVarError: ‘@fail‘ not defined in ‘Main‘
...
```

~or

```
struct OrSyntaxData <: AbstractPatternFormSyntaxData end
```

`~or` combines one or more patterns with short-circuiting logical `or` semantics. `~or` forms only have explicit notation.

```
julia> equals_x = @pattern ~or(
           x == {first},
           {second} == x
       )
Pattern:
[~or]
  [call-i]
    x                                   :: Identifier
    ==                                  :: Identifier
    first:::expr                        :: ~var
  [call-i]
    second:::expr                       :: ~var
    ==                                  :: Identifier
    x                                   :: Identifier

julia> syntax_match(equals_x, parsestmt(SyntaxNode, "x == 2"))
BindingSet with 1 entry:
  :first => Binding:
              Name: :first
              Bound source: 2 @ 1:6
              Ellipsis depth: 0
              Sub-bindings:
                BindingSet with 0 entries

julia> syntax_match(equals_x, parsestmt(SyntaxNode, "2 == x"))
BindingSet with 1 entry:
  :second => Binding:
               Name: :second
               Bound source: 2 @ 1:1
               Ellipsis depth: 0
               Sub-bindings:
                 BindingSet with 0 entries
```

~and

```
struct AndSyntaxData <: AbstractPatternFormSyntaxData end
```

`~and` combines one or more patterns with short-circuiting logical `and` semantics. Like `~or`, `~and` can only be written explicitly.

```
julia> conflicting_and = @pattern ~and({a} + 2, {a} + 3)
Pattern:
[~and]
  [call-i]
    a:::expr                            :: ~var
    +                                   :: Identifier
    2                                   :: Integer
  [call-i]
    a:::expr                            :: ~var
    +                                   :: Identifier
    3                                   :: Integer
```

```
julia> syntax_match(conflicting_and, parsestmt(SyntaxNode, "a + 2"))
MatchFail("no match")
```

~rep

```
struct RepSyntaxData <: AbstractPatternFormSyntaxData
    rep_vars::Vector{RepVar}
end
```

`~rep` denotes a pattern wrapped in one level of *ellipses*, indicating it may repeat zero or more times. Nested `~rep` forms are allowed within the enclosed pattern.

```
julia> match_all = @pattern ~rep({ex})
Pattern:
[~rep]
  ex:::expr                                 :: ~var

julia> syntax_match(match_all, parseall(SyntaxNode, """
                                                    x = 2
                                                    x + 1
                                                    """))
BindingSet with 1 entry:
  :ex => Binding:
           Name: :ex
           Bound sources: [(= x 2) @ 1:1, (call-i x + 1) @ 2:1]
           Ellipsis depth: 1
           Sub-bindings:
             [
              BindingSet with 0 entries,
              BindingSet with 0 entries
             ]
```

A `~rep` form may be represented implicitly using ellipses. The above `match_all` pattern can be rewritten as `@pattern {ex}...`.

The sugared `~rep` notation hijacks Julia's built-in splat operator. It is possible to refer to the splat operator instead of `~rep` in a pattern by escaping the pattern expression.

```
julia> @pattern v...
Pattern:
[~rep]
  v                                         :: Identifier

julia> @pattern @esc(v...)
Pattern:
[...]
  v                                         :: Identifier
```

`@esc` escapes the entire encapsulated expression by default, but supports specifying a custom escape level.

```
julia> @pattern @esc([{elems}...]...)
Pattern:
[...]
  [vect]
    [...]
      [braces]
        elems                           :: Identifier

julia> @pattern @esc([{elems}...]..., 1)
Pattern:
[...]
  [vect]
    [~rep]
      elems:::expr                      :: ~var

julia> @pattern @esc([{elems}...]..., 3)
Pattern:
[...]
  [vect]
    [...]
      elems:::expr                      :: ~var
```

Syntax Classes

`~var` forms reference syntax classes in patterns. Matching a pattern that references an unregistered syntax class causes a match time error.

```
julia> syntax_match((@pattern {x:::unregistered}), parsestmt(SyntaxNode, "dummy"))
ERROR: SyntaxClassRegistryKeyError: unregistered syntax class :unregistered
...
```

Syntax classes can be defined using the `@syntax_class` macro. They follow the behaviour described in section 2.2 for `syntax/parse`. A syntax class contains a description and a sequence of one or more patterns.

```
julia> vec = @syntax_class "vector" begin
           @pattern [{_}...]
       end
SyntaxClass: vector
  Pattern alternative #1:
    [vect]
      [~rep]
        _:::expr                        :: ~var

julia> syntax_match(vec, parsestmt(SyntaxNode, "[1, [2]]"))
BindingSet with 0 entries
```

In order to use them in patterns, syntax classes need to be registered.

```
julia> register_syntax_class!(:vec, vec);

julia> vec_of_vecs = @pattern [{v:::vec}...];

julia> syntax_match(vec_of_vecs, parsestmt(SyntaxNode, "[[1, 2], [[3]]]"))
BindingSet with 1 entry:
  :v => Binding:
          Name: :v
          Bound sources: [(vect 1 2) @ 1:2, (vect (vect 3)) @ 1:10]
          Ellipsis depth: 1
          Sub-bindings:
            [
             BindingSet with 0 entries,
             BindingSet with 0 entries
            ]
```

Argus provides a set of pre-defined syntax classes, including `vec`. The built-in implementations use `@define_syntax_class` to both define and register syntax classes at the same time. The following is Argus's implementation of the `fundef` syntax class:

```
@define_syntax_class :fundef "function definition" begin
    @pattern {call:::funcall} = {body}
    @pattern function ({call:::funcall}) {body}... end
end
```

Writing multiple patterns in a `@syntax_class` body is semantically equivalent to combining the patterns in an `~or` form.

A bound pattern variable inherits all the pattern variables bound by its constraining syntax class as sub-bindings, which can be accessed as fields.

```
@pattern begin
    {x:::assign}
    @fail x.rhs.value == 2 "rhs is two"
end
```

Comparing Patterns

The basic operation for matching syntax is the comparison of syntactic objects. In the general case, Argus considers two syntax nodes to be *compatible* with each other if:

- they have the same head;
- they contain the same value (or neither contains a value);
- they are either both leafs, or both non-leafs and the children list of one node is compatible with the children list of the other.

A `~var` node v is compatible with a `SyntaxNode` s if the syntax class constraining the pattern variable referenced by v is compatible with s. The pattern variable needs to be either unbound or bound to a `SyntaxNode` compatible with s. Matching v with s binds its pattern variable to s. A `~var` node inside a `~rep` node is compatible with a `SyntaxNode` regardless of whether its pattern variable is already bound. Pattern variable binding is handled by `~rep`.

`~fail` nodes are not compared with other nodes. A `~fail` node f passes a comparison test if all the pattern variables referenced in f's fail condition are bound and the fail condition is not satisfied.

An `~or` node o is compatible with a `SyntaxNode` s if any of o's branches is compatible with s. During matching, o records all potentially succeeding paths encountered as *recovery states*.

An `~and` node a is compatible with a `SyntaxNode` s if all a's branches are compatible with s. In case of incompatibility, the matching tries to backtrack through available recovery states. Like o, a records possible recovery states. It inherits and combines the recovery states of its branches. The combined result can be thought of as the cartesian product of the branches' recovery states.

A `~rep` node r is compatible with a `SyntaxNode` s if the node enclosed in r is compatible with s. r is compatible with a sequence of `SyntaxNode`s ss if ss is empty or if the node enclosed in r is compatible with all the nodes in ss. Matching r binds pattern variables to sequences of nodes. Like o and a, r tracks recovery states. During the matching of ss with a `~rep` node rc contained in a complex pattern, rc greedily consumes all the nodes it can by default. It is possible to change the behaviour to a non-greedy one.

```
julia> ones_vec = @pattern [1, {ones1}..., 1, {ones2}...];

julia> syntax_match(ones_vec, parsestmt(SyntaxNode, "[1, 1, 1, 1]"))
BindingSet with 2 entries:
  :ones2 => Binding:
              Name: :ones2
              Bound sources: []
              Ellipsis depth: 1
              Sub-bindings:
                []
  :ones1 => Binding:
              Name: :ones1
              Bound sources: [1 @ 1:5, 1 @ 1:8]
              Ellipsis depth: 1
              Sub-bindings:
                [
                 BindingSet with 0 entries,
                 BindingSet with 0 entries
                ]
```

```
julia> syntax_match(ones_vec, parsestmt(SyntaxNode, "[1, 1, 1, 1]"); greedy=false)
BindingSet with 2 entries:
  :ones2 => Binding:
              Name: :ones2
              Bound sources: [1 @ 1:8, 1 @ 1:11]
              Ellipsis depth: 1
              Sub-bindings:
                [
                 BindingSet with 0 entries,
                 BindingSet with 0 entries
                ]
  :ones1 => Binding:
              Name: :ones1
              Bound sources: []
              Ellipsis depth: 1
              Sub-bindings: []
```

Only `~var` and `~rep` nodes can bind pattern variables.

3.2.2 Writing Rules

A rule is specified by its name, description and pattern.

```
struct Rule
    name::String
    description::String
    pattern::Pattern
end
```

A rule match recursively traverses a given unit of source code and collects the results of matching sub-expressions with the rule's pattern. Bound pattern variables are returned in corresponding binding sets [24].

```
julia> assignments = @rule "assignments" begin
           description = "Searching for assignments."
           pattern = @pattern {a:::assign}
       end
assignments: Searching for assignments.
Pattern:
a:::assign                                :: ~var

julia> src = """
       x = 2
       y = x + 1
       if y == x
           z = true
       end
       f() = 4
       """;
```

```
julia> match_result = rule_match(assignments, parseall(SyntaxNode, src))
MatchResults with 3 matches and 0 failures:
Matches:
  BindingSet(:a => Binding(:a,
                           (= x 2) @ 1:1,
                           BindingSet(:rhs => Binding(:rhs, 2 @ 1:5, BindingSet()),
                                      :lhs => Binding(:lhs,
                                                      x @ 1:1,
                                                      BindingSet(:_id => ...)))))
  BindingSet(:a => Binding(:a,
                           (= y (call-i x + 1)) @ 2:1,
                           BindingSet(:rhs => Binding(:rhs,
                                                      (call-i x + 1) @ 2:5,
                                                      BindingSet()),
                                      :lhs => Binding(:lhs,
                                                      y @ 2:1,
                                                      BindingSet(:_id => ...)))))
  BindingSet(:a => Binding(:a,
                           (= z true) @ 4:5,
                           BindingSet(:rhs => Binding(:rhs,
                                                      true @ 4:9,
                                                      BindingSet()),
                                      :lhs => Binding(:lhs,
                                                      z @ 4:5,
                                                      BindingSet(:_id => ...)))))
```

Rules can be organised into *rule groups*. A rule group is a dictionary mapping rule names to their associated definitions.

```
struct RuleGroup <: AbstractDict{String, Rule}
    name::String
    rules::Dict{String, Rule}
end
```

Defined rules can be added to a rule group using `register_rule!`. Alternatively, a rule can be defined and registered in a group at the same time using `@define_rule_in_group`.

```
lang = RuleGroup("lang")

@define_rule_in_group lang "chained-const-assignment" begin
    description = """
    Do not chain assignments with const. The right hand side is not constant here.
    """

    pattern = @pattern begin
        const {_:::identifier} = {_:::identifier} = {_}
    end
end
```

3.3 Step-by-Step Examples

3.3.1 Combined Patterns

Consider the following `~or` pattern match:

```
julia> pattern = @pattern begin
           ~or(
               {x:::literal},
               {x:::identifier}
           )
       end;

julia> src = parsestmt(SyntaxNode, "id");

julia> syntax_match(pattern, src)
BindingSet with 1 entry:
  :x => Binding:
          Name: :x
          Bound source: id @ 1:1
          Ellipsis depth: 0
          Sub-bindings:
            BindingSet with 1 entry:
              :_id => Binding:
                        Name: :_id
                        Bound source: id @ 1:1
                        Ellipsis depth: 0
                        Sub-bindings:
                          BindingSet with 0 entries
```

First, `syntax_match` detects that the pattern is an `~or` form and dispatches the matching to `syntax_match_or`. The `~or` branches are tried one by one.

The first branch and the source node are passed to `syntax_match`[8]. This step conceptually corresponds to the following call:

```
syntax_match({x:::literal}, id)
```

Or, using the explicit notation:

```
syntax_match(~var(:x, :literal), id)
```

The matching is delegated to `syntax_match_var`, which searches for the syntax class associated with the name `literal` in the syntax class registry. It finds the built-in definition:

[8]They are actually passed to a helper function `_syntax_match` which is responsible for the core match logic.

```
@syntax_class "literal" begin
    @pattern begin
        {_lit}
        @fail begin
            using JuliaSyntax: is_literal
            !is_literal(_lit.src)
        end ""
    end
end
```

Next, `syntax_match_var` attempts to match `literal` with the source node `id` by going through the patterns in its definition one by one. The first (and only) pattern to match is:

```
@pattern begin
    {_lit}
    @fail begin
        using JuliaSyntax: is_literal
        !is_literal(_lit.src)
    end ""
end
```

This expands to an `~and` pattern, as seen in the section describing `~fail`. Therefore, the following match is performed:

```
syntax_match(~and({_lit},
                  ~fail(begin
                            using JuliaSyntax: is_literal
                            !is_literal(_lit.src)
                        end,
                        "")),
             id)
```

Since the pattern is an `~and` form, the matching is handled by `syntax_match_and`. Like `syntax_match_or`, `syntax_match_and` goes through its branches one by one and attempts to match them with the source node. The first branch is `{_lit}`:

```
syntax_match(~var(:_lit, :expr),
             id)
```

`syntax_match_var` takes over and searches for the definition of `expr`, which is the most fundamental built-in syntax class:

```
@syntax_class "expr" begin
    @pattern ~fail(false, "")
end
```

Matching `expr` never fails. Therefore, `syntax_match_var` successfully matches `expr` with `id`. Next, it looks for `_lit` in the bindings gathered so far. There are no bindings, which means that `_lit` can be safely bound to `id`. A successful match is returned, with the new binding added to the final binding set. This marks the end of the matching for the first `~and` branch.

The next branch is a `~fail` form, handled by `syntax_match_fail`. No comparison is necessary inside `syntax_match_fail`. Instead, the node's fail condition is evaluated using the accumulated bindings. Conceptually, this corresponds to performing the following function call:

```
function fail_condition(binding_set)
    using JuliaSyntax: is_literal
    return !is_literal(binding_set[:_lit].src)
end

fail_condition(BindingSet(:_lit => id))
```

Because `id` is an identifier and not a literal, the fail condition returns `true`. The matching therefore fails.

Matching the second `~and` branch fails. `syntax_match_and` tries to recover, but there are no recovery states marked by its previous branches. Therefore, it cannot recover and the entire `~and` match fails.

We are now at the end of the match for the `literal` syntax class. Since none of its patterns provided a successful match, there can be no match for the syntax class altogether. Therefore, `syntax_match` fails with the message `"expected literal"`. The string after `"expected "` comes from the syntax class's description.

We are back to matching `~or`:

```
syntax_match_or(~or({x:::literal},
                    {x:::identifier}
                    ),
                id)
```

If matching the first branch had succeeded, `syntax_match_or` would have recorded `~or({x:::identifier})` as a possible recovery state. Surrounding patterns would have then been able to try another path if they failed to match on the current one. But since it failed, the matching directly continues with `{x:::identifier}`:

```
syntax_match_var(~var(:x, :identifier), id)
```

The steps are the same as for matching `{x:::literal}`. The only difference is the definition of the syntax class:

```
@syntax_class "identifier" begin
    @pattern begin
        {_id}
        @fail begin
            using JuliaSyntax: is_identifier
            !is_identifier(_id.src)
        end ""
    end
end
```

This time, the syntax class matches and returns the binding of the pattern variable `_id` to the source node `id`. `syntax_match_var` stores it as a sub-binding of the pattern variable `x` and binds `x` to `id` as well.

This `~or` branch thus reports a successful match. `syntax_match_or` still does not record a recovery state, this time because `{x:::identifier}` is the last branch in the pattern.

Since `syntax_match_or` found a matching branch it can return the success back to `syntax_match`, which returns the final result:

```
BindingSet with 1 entry:
  :x => Binding:
          Name: :x
          Bound source: id @ 1:1
          Ellipsis depth: 0
          Sub-bindings:
            BindingSet with 1 entry:
              :_id => Binding:
                        Name: :_id
                        Bound source: id @ 1:1
                        Ellipsis depth: 0
                        Sub-bindings:
                          BindingSet with 0 entries
```

3.3.2 Patterns with Repetitions

A `~rep` match follows a different algorithm than the one described earlier. We will study it using the `ones_vec` pattern defined in the section on Comparing Patterns.

```
julia> ones_vec = @pattern [1, {ones1}..., 1, {ones2}...];

julia> syntax_match(ones_vec, parsestmt(SyntaxNode, "[1, 1, 1, 1]"))
BindingSet with 2 entries:
  :ones2 => Binding:
              Name: :ones2
              Bound sources: []
              Ellipsis depth: 1
              Sub-bindings:
                []
```

```
  :ones1 => Binding:
              Name: :ones1
              Bound sources: [1 @ 1:5, 1 @ 1:8]
              Ellipsis depth: 1
              Sub-bindings:
                [
                 BindingSet with 0 entries,
                 BindingSet with 0 entries
                ]
```

This time, `syntax_match` notices that the pattern is *not* a special form. The pattern's head and value are the same as the source node's — `vect` and `nothing`. Therefore, `syntax_match` proceeds to compare the children lists:

```
pattern_children = [
    1,
    {ones1}...,
    1,
    {ones2}...
]

src_children = [
    1,
    1,
    1,
    1
]
```

The children are compared with each other, starting with `1` from `pattern_children` and `1` from `src_children`. The match is trivial and results in a success with no new bindings.

The next pair is `{ones1}...` and `1`. The matching is dispatched to `syntax_match_rep`, which first tries to match `{ones1}` with `1`. Since `{ones1}` is unconstrained[9], the matching succeeds and `syntax_match_rep` can bind the pattern variable. `syntax_match_rep` first checks whether `ones1` is already bound, just as `syntax_match_var` would do. Seeing that it isn't, it proceeds to bind it. However, instead of binding `{ones1}` to the single node `1`, it binds it to a sequence of nodes containing `1` as its only element.

Matching `{ones1}...` with `1` thus succeeds, producing a binding for `ones1`. Here, a recovery state needs to be marked in case the path we choose to continue with fails to match. The path choice depends on whether the matching algorithm is greedy or not.

In a greedy algorithm, a `~rep` node consumes as many nodes as possible. In our example, the preferred path is that where `{ones1}...` consumes `1`. The matching continues with the same pattern sequence — `[{ones1}..., 1, {ones2}...]` —, the remaining

[9] A node constrained by `expr` can be considered unconstrained.

source node sequence — `[1, 1]` — and the binding `:ones1 => [1]`. The alternative path is that where `{ones1}...` does not consume `1`. The recovery state therefore contains the patterns `[1, {ones2}...]`, the source nodes `[1, 1, 1]` and the binding `:ones1 => []`.

For a non-greedy algorithm, the preferred and alternative paths are reversed. Since this is the only difference between greedy and non-greedy, we will continue by discussing only the former.

Continuing on the greedy preferred path, the matching resumes with the following information:

```
pattern_children = [
    {ones1}...,
    1,
    {ones2}...
]

src_children = [
    1,
    1
]

BindingSet(:ones1 => [1])

recovery_stack = [
    ([1, {ones2}...], [1, 1, 1], BindingSet(:ones1 => []))
]
```

`{ones1}...` continues to consume source nodes until there are none left. Each consumed node is added to `ones1`'s list of bound nodes:

```
pattern_children = [
    {ones1}...,
    1,
    {ones2}...
]

src_children = []

BindingSet(:ones1 => [1, 1, 1])

recovery_stack = [
    ([1, {ones2}...], [1, 1, 1], BindingSet(:ones1 => [])),
    ([1, {ones2}...], [1, 1], BindingSet(:ones1 => [1])),
    ([1, {ones2}...], [1], BindingSet(:ones1 => [1, 1]))
]
```

The matching continues after removing `{ones1}...` from the pattern list. Next, the pattern `1` should be compared to the next source node, but there are none left. This

means there can be no match on this path. The most recent alternative path is tried instead:

```
pattern_children = [
    1,
    {ones2}...
]

src_children = [1]

BindingSet(:ones1 => [1, 1])

recovery_stack = [
    ([1, {ones2}...], [1, 1, 1], BindingSet(:ones1 => [])),
    ([1, {ones2}...], [1, 1], BindingSet(:ones1 => [1]))
]
```

On this path, the pattern `1` can be matched with the source node `1` and the matching can continue with the patterns `[{ones2}...]` and the source nodes `[]`.

`syntax_match_rep` is called for matching `{ones2}...` with the empty source nodes sequence. The result contains the new binding `:ones2 => []`:

```
BindingSet(:ones1 => [1, 1],
           :ones2 => [])
```

There are no more source nodes to match, but now there are no more patterns either. The match is therefore successful and `syntax_match` returns the final result:

```
BindingSet with 2 entries:
  :ones2 => Binding:
              Name: :ones2
              Bound sources: []
              Ellipsis depth: 1
              Sub-bindings:
                []
  :ones1 => Binding:
              Name: :ones1
              Bound sources: [1 @ 1:5, 1 @ 1:8]
              Ellipsis depth: 1
              Sub-bindings:
                [
                 BindingSet with 0 entries,
                 BindingSet with 0 entries
                ]
```

3.3.3 Rules

A rule match exhausts all the possibilities of a pattern match. A successful pattern match does not immediately return. The result is instead stored in a collection of results. The

matching then continues with all the recovery states recorded during the pattern match. After the recovery states are completely explored, the pattern is recursively matched against all the source node's children.

We will examine the vector example from the previous section in the context of rules:

```
julia> ones_rule = @rule "ones" begin
           description = "An array containing at least two ones."
           pattern = @pattern [1, {ones1}..., 1, {ones2}...]
       end;

julia> rule_match(ones_rule, parsestmt(SyntaxNode, "[1, 1, 1, 1]"))
MatchResults with 3 matches and 0 failures:
Matches:
  BindingSet(:ones1 => Binding(:ones1,
                               [1 @ 1:5, 1 @ 1:8],
                               BindingSet[BindingSet(), BindingSet()]),
             :ones2 => Binding(:ones2,
                               [],
                               BindingSet[]))
  BindingSet(:ones1 => Binding(:ones1,
                               [1 @ 1:5],
                               BindingSet[BindingSet()]),
             :ones2 => Binding(:ones2,
                               [1 @ 1:11],
                               BindingSet[BindingSet()]))
  BindingSet(:ones1 => Binding(:ones1,
                               [],
                               BindingSet[]),
             :ones2 => Binding(:ones2,
                               [1 @ 1:8, 1 @ 1:11],
                               BindingSet[BindingSet(), BindingSet()]))
```

`rule_match` sends the rule's pattern and the source node to `syntax_match`[10]. The matching follows the steps outlined in section 3.3.2, producing the first rule match result:

```
BindingSet(:ones1 => [1, 1],
           :ones2 => [])

recovery_stack = [
    ((vect 1 {ones2}...), (vect 1 1), BindingSet(:ones1 => [1])),
    ((vect 1 {ones2}...), (vect 1 1 1), BindingSet(:ones1 => []))
]
```

The recovery states gathered when matching the children lists are propagated to the parent match. `rule_match` stores the resulted `BindingSet` and continues by matching the first recovery state:

[10]The matching is actually handled by `syntax_match_all`, but discussing this extra layer adds complexity without providing meaningful insight.

```
syntax_match((vect 1 {ones2}...), (vect 1 1), BindingSet(:ones => [1]))

recovery_stack = [
    ((vect 1 {ones2}...), (vect 1 1 1), BindingSet(:ones1 => []))
]
```

Following the same algorithm, the matching is successful and produces a new binding set:

```
BindingSet(:ones1 => [1],
           :ones2 => [1])
```

Only one recovery state remains:

```
syntax_match((vect 1 {ones2}...), (vect 1 1 1), BindingSet(:ones => []))

recovery_stack = []
```

This matching succeeds as well, adding one more binding set to the final result:

```
BindingSet(:ones1 => [],
           :ones2 => [1, 1])
```

The overall rule match returns three successful match results:

```
MatchResults with 3 matches and 0 failures:
Matches:
  BindingSet(:ones1 => Binding(:ones1,
                               [1 @ 1:5, 1 @ 1:8],
                               BindingSet[BindingSet(), BindingSet()]),
             :ones2 => Binding(:ones2,
                               [],
                               BindingSet[]))
  BindingSet(:ones1 => Binding(:ones1,
                               [1 @ 1:5],
                               BindingSet[BindingSet()]),
             :ones2 => Binding(:ones2,
                               [1 @ 1:11],
                               BindingSet[BindingSet()]))
  BindingSet(:ones1 => Binding(:ones1,
                               [],
                               BindingSet[]),
             :ones2 => Binding(:ones2,
                               [1 @ 1:8, 1 @ 1:11],
                               BindingSet[BindingSet(), BindingSet()]))
```

3.4 Real-World Scenario

An audit event is a record of a security-relevant occurrence in a system [25]. It is useful for detecting system misuse or analysing the system's behaviour. A company might want to keep track of the audit events in their codebase for many reasons, including law enforcement.

Suppose a company with a Julia codebase wants to log all audit events in a file. In their case, an audit event is implemented as an `@audit_event` macro with the following signature:

```
audit_event(event::Symbol, resource::Symbol, other_args...)
```

The company wants their logs to have the following structure:

```
File: <file_name>
Calling function: <function_name>
Call location: (<line>, <column>)
Event: <event_name>
Resource: <resource_type>

File: ...
...
```

The task can be approached in many ways. A regex could be the first choice, but ensuring that it catches all `@audit_event` calls might be difficult. Moreover, regexes cannot provide information about file or source location. Custom parsing of the target files would be similarly challenging.

Argus provides an elegant and powerful solution to this problem:

```
@define_syntax_class :audit_event_call "@audit_event call" begin
    @pattern begin
        {m:::macrocall}
        @fail m.name != "@audit_event" ""
    end
end

@define_syntax_class :audit_event_log "@audit_event call log" begin
    @pattern begin
        {audit_event:::audit_event_call}
        @fail begin
            using JuliaSyntax: SyntaxNode, Kind, kind, source_location, sourcefile

            function find_calling_function(src::SyntaxNode)
                if kind(src) === Kind("function")
                    call_node = src.children[1]
                    function_name_node = call_node.children[1]
                    return function_name_node.val
```

```
                end
                return find_calling_function(src.parent)
            end

            # The first two arguments should be symbols. Symbols are syntactically
            # represented as:
            #
            # [quote-:]
            #   <symbol_name>
            args = map(symbol_node -> symbol_node.children[1].val,
                       audit_event.m.args[1:2])

            src = audit_event.src
            println("File: ", sourcefile(src).filename)
            println("Calling function: ", find_calling_function(src))
            println("Call location: ", source_location(src))
            println("Event: ", args[1])
            println("Resource: ", args[2])
            println()

            false  # `@fail` return value.
        end ""
    end
end

audit_event_rule = @rule "@audit_event_logs" begin
    description = "Log @audit_event calls."
    pattern = @pattern {a::audit_event_log}
end
```

The `audit_event_call` syntax class matches `@audit_event` macro calls with any number of arguments. `audit_event_log` is a logger syntax class that matches and prints information about an `audit_event_call`. The final rule simply matches this logger syntax class.

Suppose an input file `"audit.jl"` has the following content:

```
function do_things_with_a_user(user::User, group::Group, add=false)
    create_user(user)
    @audit_event(:created, :user, user.id)

    if add
        add_user(user, group)
        @audit_event(:added, :user, user.id, group.id)
    end
end

function do_things_with_a_group(group::Group)
    make_public(group)
    @audit_event(:modified, :group, group.id, :public)
end
```

We can test our rule against this input file.

```
julia> rule_match(audit_event_rule, "audit.jl");
File: audit.jl
Calling function: do_things_with_a_user
Call location: (3, 5)
Event: created
Resource type: user

File: audit.jl
Calling function: do_things_with_a_user
Call location: (7, 9)
Event: added
Resource type: user

File: audit.jl
Calling function: do_things_with_a_group
Call location: (13, 5)
Event: modified
Resource type: group
```

The output can be easily redirected to a log file:

```
open("audit_logs.txt", "w") do io
    redirect_stdout(io) do
        rule_match(audit_event_rule, "audit.jl")
    end
end
```

A similar result can be obtained with MacroTools.jl [9], using `postwalk` to recurse into sub-expressions and `@capture` to bind pattern variables:

```
using MacroTools: postwalk, @capture
using JuliaSyntax: parseall

function audit_event_log(filename::String)
    ex = parseall(Expr, read(filename, String); filename=filename)

    postwalk(ex) do x
        if @capture(x, function f_(args__) body_ end)
            println("File: ", body.args[1].file)
            println("Function: ", f)
            postwalk(body) do y
                if @capture(y, @m_(ev_, res_, args__)) && m == Symbol("@audit_event")
                    @capture(y, macrocall_) &&
                        println("Calling from: ", macrocall.args[2].line)
                    println("Event: ", ev)
                    println("Resource: ", res)
                    println()
                else
```

```
                    return y
                end
            end
        else  # Not a match for the function definition.
            return x
        end
    end
end
```

The result is essentially the same as that of `rule_match`:

```
julia> audit_event_log("audit.jl");
File: audit.jl
Function: do_things_with_a_user
Calling from: 3
Event: :created
Resource: :user

Calling from: 7
Event: :added
Resource: :user

File: audit.jl
Function: do_things_with_a_group
Calling from: 13
Event: :modified
Resource: :group
```

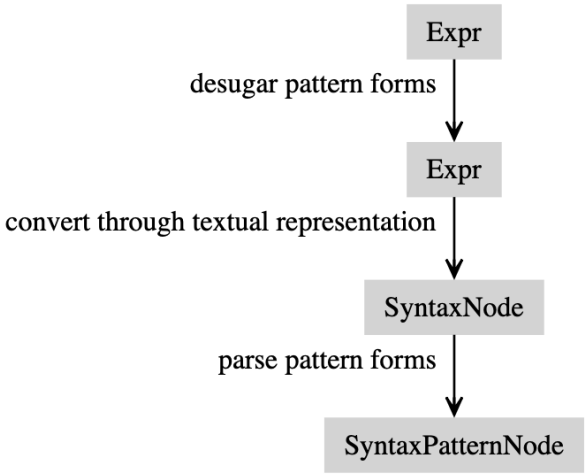
While this approach results in less code, it is more difficult to understand for the less-experienced Julia programmer. Moreover, it cannot provide column information because of the reasons outlined in section 3.1.

Argus's main advantage over MacroTools is its extensibility: `audit_event_call` is defined as a syntax class and hence can be easily reused in other patterns and rules — for example to generate other logs or analyse surrounding expressions.

4 Limitations and Future Steps

Argus already lays out a powerful and expressive language for program analysis. However, additional efforts are needed to fully realise its potential.

The current design of `SyntaxPatternNode` suffers from a structural weakness associated with `Expr` parsing. The transformation from `Expr` to `SyntaxPatternNode` passes through `SyntaxNode` in roughly the following way:



The flaw is introduced in the `Expr` → `SyntaxNode` step. Converting an `Expr` to a `SyntaxNode` based on textual representation is not always reliable. Though few in number, certain differences exist between the two representations [26]. A notable disparity concerns implicit `block` nodes. `Expr` includes extra `block`s in order to attach `LineNumberNodes` to its children. This is the case for a short form function definition:

```
julia> :( f(x) = 2 )
:(f(x) = begin
          #= REPL[16]:1 =#
          2
      end)
```

`SyntaxNode`s inherently store source location information and hence don't require extra nodes:

```
julia> JuliaSyntax.parsestmt(SyntaxNode, "f(x) = 2")
SyntaxNode:
[function-=]
  [call]
    f                                   :: Identifier
    x                                   :: Identifier
  2                                     :: Integer
```

Argus uses the `Expr` representation for parsing and the `SyntaxNode` one for matching. This causes unexpected behaviour when the representations differ:

```
julia> syntax_match((@pattern f(x) = 2), parsestmt(SyntaxNode, "f(x) = 2"))
MatchFail("no match")

julia> syntax_match((@pattern f(x) = 2), parsestmt(SyntaxNode, "f(x) = begin 2 end"))
BindingSet with 0 entries
```

A robust translation from `Expr` to `SyntaxNode` would solve this problem.

The syntax matching mechanism could be extended to support templates. This would allow Argus to serve as a tool for writing macros, aligning it more closely with `syntax/parse`. The rule framework could then be improved as well by adding suggestions and code refactoring.

Another limitation comes from Argus's pattern fail conditions. A satisfied fail condition causes a match failure. It is not possible to report a source node that fits the superficial layout of a pattern, but does not meet certain criteria. The value of *late failures* is demonstrated by the `let` example from section 2.1:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let ([var:identifier rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate #'(var ...))
                 "duplicate variable name"
     #'((lambda (var ...) body) rhs ...)]))

(let ([a 1] [b 2]) (+ a b))
  compile-time expansion: ((lambda (a b) (+ a b)) 1 2)
  run-time result: 3
(let (["a" 1]) (add1 a))
  compile-time error: expected identifier
(let ([a 1] [a 2]) (+ a a))
  compile-time error: duplicate variable name
```

The goal is for a `let` definition to match the pattern `(let ([var:identifier rhs:expr] ...) body:expr)` and *report an error* if there are duplicate bindings. Argus would simply return no match in case of misuse of a structurally matching source node, just as it would for a structurally unrelated source node. Late failures are especially relevant for writing macros.

`syntax/parse` sets a noteworthy precedent in reporting errors [16]. Argus could follow the example and learn to provide more specific failure messages.

User-oriented improvements include compiling a default set of static analysis rules and offering a command line interface. The latter depends on current possibilities for ahead-of-time compilation in Julia [27].

5 Conclusions

We have shown that Julia can learn from languages with advanced metaprogramming support in order to provide powerful tools for examining and manipulating syntax. We introduced Argus.jl as a more expressive alternative to other existing Julia static analysis packages. Its core concepts and driving algorithms have been discussed at length. All underlying mechanisms have been illustrated through step-by-step examples.

We have proven Argus's practical applicability in a real-world use case. Its flexibility was demonstrated by its ability to successfully match source code while gathering and logging information in a specified format.

In the future, Argus could be more tightly integrated with Julia's metaprogramming system. It could continue to draw inspiration from other programming languages in order to select and report specific errors in case of misuse. Its rule writing engine could be presented as a command line tool that scans files and refactors code. Argus could expose a set of default static analysis rules readily available for use.

6 Bibliography

References:

[1] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM*, vol. 3, pp. 184–195, 1960, HTTPS://DL.ACM.ORG/DOI/10.1145/367177.367199

[2] S. Karpinski, "In what sense are languages like Elixir and Julia homoiconic?" 2015. HTTPS://STACKOVERFLOW.COM/A/31734725. [Accessed: Jun. 15, 2025]

[3] "Julia Documentation — Metaprogramming." HTTPS://DOCS.JULIALANG.ORG/EN/V1/MANUAL/METAPROGRAMMING/. [Accessed: Jun. 15, 2025]

[4] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Why We Created Julia," 2012, HTTPS://JULIALANG.ORG/BLOG/2012/02/WHY-WE-CREATED-JULIA/

[5] JuliaLang, "JuliaSyntax.jl — Parsing with ParseStream." HTTPS://JULIALANG.GITHUB.IO/JULIASYNTAX.JL/DEV/DESIGN/#PARSING-WITH-PARSESTREAM. [Accessed: Jun. 15, 2025]

[6] JuliaLang, "JuliaSyntax.jl — Syntax Trees." HTTPS://JULIALANG.GITHUB.IO/JULIASYNTAX.JL/DEV/REFERENCE/. [Accessed: Jun. 15, 2025]

[7] S. Kadowaki, "JET.jl — An experimental code analyzer for Julia." HTTPS://GITHUB.COM/AVIATESK/JET.JL. [Accessed: Jun. 15, 2025]

[8] JuliaDebug, "Cthulhu.jl — The slow descent into madness." HTTPS://GITHUB.COM/JULIADEBUG/CTHULHU.JL. [Accessed: Jun. 15, 2025]

[9] FluxML, "MacroTools.jl — A library of tools for working with Julia code and expressions." HTTPS://GITHUB.COM/FLUXML/MACROTOOLS.JL. [Accessed: Jun. 15, 2025]

[10] T. Zhao, "MLStyle.jl — Julia functional programming infrastructures and metaprogramming facilities." HTTPS://GITHUB.COM/THAUTWARM/MLSTYLE.JL. [Accessed: Jun. 15, 2025]

[11] M. Felleisen *et al.*, "The Racket Manifesto," *First Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32, pp. 113–128, 2015, HTTPS://DOI.ORG/10.4230/LIPICS.SNAPL.2015.113

[12] "Racket Documentation — Parsing and Specifying Syntax." HTTPS://DOCS.RACKET-LANG.ORG/SYNTAX/STXPARSE.HTML. [Accessed: Jun. 16, 2025]

[13] "Racket Documentation — Specifying Syntax with Syntax Classes." HTTPS://DOCS.RACKET-LANG.ORG/SYNTAX/STXPARSE-SPECIFYING.HTML. [Accessed: Jun. 16, 2025]

[14] "Racket Documentation — Resyntax." https://docs.racket-lang.org/resyntax/index.html. [Accessed: Jun. 16, 2025]

[15] E. E. Kohlbecker and M. Wand, "Macro-by-Example: Deriving Syntactic Transformations from their Specifications," *ACM Symposium on Principles of Programming Languages*, pp. 77–84, 1986, https://doi.org/10.1145/41625.41632

[16] R. Culpepper, "Fortifying Macros," *Journal of Functional Programming*, vol. 22, no. 4-5, pp. 439–476, 2012, DOI:10.1017/S0956796812000275

[17] "Julia Documentation — Developer Documentation — Julia ASTs." https://docs.julialang.org/en/v1/devdocs/ast/#Julia-ASTs. [Accessed: Jun. 17, 2025]

[18] "Julia Documentation — The Julia REPL." https://docs.julialang.org/en/v1/stdlib/REPL/. [Accessed: Jun. 17, 2025]

[19] "JuliaSyntax.jl." https://julialang.github.io/JuliaSyntax.jl/dev/. [Accessed: Jun. 17, 2025]

[20] "JuliaSyntax.jl — More about syntax kinds." https://julialang.github.io/JuliaSyntax.jl/dev/design/#More-about-syntax-kinds. [Accessed: Jun. 17, 2025]

[21] "Domain-specific language." https://en.wikipedia.org/wiki/Domain-specific_language. [Accessed: Jun. 17, 2025]

[22] "Racket Documentation — Syntax Patterns." https://docs.racket-lang.org/syntax/stxparse-patterns.html. [Accessed: Jun. 17, 2025]

[23] "Julia Documentation — Compound Expressions." https://docs.julialang.org/en/v1/manual/control-flow/#man-compound-expressions. [Accessed: Jun. 19, 2025]

[24] I. Dumitru, "Argus.jl — Overview." https://github.com/iuliadmtru/Argus.jl/tree/5b2ab22708941453fc561f535e63f5b22b0994dc?tab=readme-ov-file#overview. [Accessed: Jun. 19, 2025]

[25] "IBM Documentation — Audit Events." https://www.ibm.com/docs/en/aix/7.3.0?topic=selection-audit-events. [Accessed: Jun. 19, 2025]

[26] "JuliaSyntax.jl — JuliaSyntax trees vs Expr." https://julialang.github.io/JuliaSyntax.jl/dev/reference/#JuliaSyntax-trees-vs-Expr. [Accessed: Jun. 19, 2025]

[27] "Julia Documentation — Ahead of Time Compilation." https://docs.julialang.org/en/v1/devdocs/aot/. [Accessed: Jun. 19, 2025]