Slang's Query API

A Better Way to Analyze Solidity Code

NOMIC
FOUNDATION

Slang's Query API

A Better Way to Analyze Solidity Code

NOMIC
FOUNDATION

What is Slang?

Slang is a modular Solidity toolkit that powers developer tools and code analysis

➔ **Purpose**: Makes it easy to analyze and work with Solidity source code
➔ **Target Users**: Tool developers, security researchers, IDE creators
➔ **Key Feature**: Powerful Query API for code analysis

Resources

➔ **GitHub**: github.com/NomicFoundation/slang
➔ **Docs**: nomicfoundation.github.io/slang/latest/
➔ **Community**: t.me/+ild4-RjaDqgxNjEx

What is Slang?

Slang is a modular Solidity toolkit that powers developer tools and code analysis

- ➔ **Purpose**: Makes it easy to analyze and work with Solidity source code
- ➔ **Target Users**: Tool developers, security researchers, IDE creators
- ➔ **Key Feature**: Powerful Query API for code analysis

Resources

- ➔ **GitHub**: github.com/NomicFoundation/slang
- ➔ **Docs**: nomicfoundation.github.io/slang/latest/
- ➔ **Community**: t.me/+ild4-RjaDqgxNjEx

Code Analysis Fundamentals

Before diving into queries, let's understand two key concepts:

Concrete Syntax Tree (CST)

➔ A complete tree representation of source code
➔ Preserves every detail including formatting and comments
➔ Like having the full book with every word and punctuation mark

Abstract Syntax Tree (AST)

➔ A simplified tree representation of code structure
➔ Focuses on essential meaning, drops formatting details
➔ Like a book summary that keeps the plot but drops the prose

Code Analysis Fundamentals

Before diving into queries, let's understand two key concepts:

Concrete Syntax Tree (CST)

- ➔ A complete tree representation of source code
- ➔ Preserves every detail including formatting and comments
- ➔ Like having the full book with every word and punctuation mark

Abstract Syntax Tree (AST)

- ➔ A simplified tree representation of code structure
- ➔ Focuses on essential meaning, drops formatting details
- ➔ Like a book summary that keeps the plot but drops the prose

Concrete Syntax Trees (CST)

```
uint x = 1;
```

CST Structure

```
VariableDeclaration
├── TypeName: "uint"
├── Trivia.Whitespace: " "
├── Identifier: "x"
├── Trivia.Whitespace: " "
├── Punctuation.Equals: "="
├── Trivia.Whitespace: " "
├── NumberLiteral: "1"
└── Punctuation.Semicolon: ";"
```

Concrete Syntax Trees (CST)

```
uint x = 1;
```

CST Structure

```
VariableDeclaration
├── TypeName: "uint"
├── Trivia.Whitespace: " "
├── Identifier: "x"
├── Trivia.Whitespace: " "
├── Punctuation.Equals: "="
├── Trivia.Whitespace: " "
├── NumberLiteral: "1"
└── Punctuation.Semicolon: ";"
```

Concrete Syntax Trees (CST)

```
uint x = 1;
```

CST Structure

```
VariableDeclaration
├── TypeName: "uint"
├── Trivia.Whitespace: " "
├── Identifier: "x"
├── Trivia.Whitespace: " "
├── Punctuation.Equals: "="
├── Trivia.Whitespace: " "
├── NumberLiteral: "1"
└── Punctuation.Semicolon: ";"
```

CST in Practice

```
function hello() {
    return "Hi!";
}
```

CST Structure

```
FunctionDefinition
├── Punctuation.Keyword: "function"
├── Identifier: "hello"
├── ParameterList: "()"
└── Block
    └── ReturnStatement
        ├── Punctuation.Keyword: "return"
        ├── StringLiteral: "Hi!"
        └── Punctuation.Token: ";"
```

CST in Practice

```
function hello() {
    return "Hi!";
}
```

CST Structure

```
FunctionDefinition
├── Punctuation.Keyword: "function"
├── Identifier: "hello"
├── ParameterList: "()"
└── Block
    └── ReturnStatement
        ├── Punctuation.Keyword: "return"
        ├── StringLiteral: "Hi!"
        └── Punctuation.Token: ";"
```

Why Concrete?

CST Preserves:

- ➔ All tokens
- ➔ Whitespace
- ➔ Comments
- ➔ Formatting
- ➔ Exact source structure

Why Concrete?

CST Preserves:

- ➔ All tokens
- ➔ Whitespace
- ➔ Comments
- ➔ Formatting
- ➔ Exact source structure

Why Abstract?

AST Omits:

- ➔ Non-essential tokens
- ➔ Formatting details
- ➔ Some parentheses
- ➔ Exact token sequence

Why Concrete?

CST Preserves:

- ➔ All tokens
- ➔ Whitespace
- ➔ Comments
- ➔ Formatting
- ➔ Exact source structure

Why Abstract?

AST Omits:

- ➔ Non-essential tokens
- ➔ Formatting details
- ➔ Some parentheses
- ➔ Exact token sequence

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

Traditional CST Navigation

```typescript
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

➔ Verbose traversal logic

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

➔ Verbose traversal logic
➔ Error prone - especially edge cases

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

- ➔ Verbose traversal logic
- ➔ Error prone - especially edge cases
- ➔ Hard to maintain

Traditional CST Navigation

```
function findVariables(node: CstNode) {
  if (node.kind === NonterminalKind.VariableDeclaration) {
    // Have to check children manually
    for (const child of node.children) {
      if (child.kind === NonterminalKind.TypeName) {
        // Process type name...
      }
      if (child.kind === TerminalKind.Identifier) {
        // Process identifier...
      }
    }
  }
  // Recursively check all children
  for (const child of node.children) {
    findVariables(child);
  }
}
```

➔ Verbose traversal logic
➔ Error prone - especially edge cases
➔ Hard to maintain

Common Challenges with Manual Traversal

```
function findAssignments(node: CstNode) {
  if (node.kind === NonterminalKind.Assignment) {
    // Need to handle parenthesized expressions
    let target = node.left;
    while (target.kind === NonterminalKind.ParenthesizedExpression) {
      target = target.expression;
    }
    // Need to handle qualified names
    if (target.kind === TerminalKind.Identifier
        || target.kind === NonterminalKind.MemberAccess) {
      // More complex logic...
    }
  }
  // Don't forget to recurse!
  for (const child of node.children) {
    findAssignments(child);
  }
}
```

➔ Recursion management

Common Challenges with Manual Traversal

```
function findAssignments(node: CstNode) {
  if (node.kind === NonterminalKind.Assignment) {
    // Need to handle parenthesized expressions
    let target = node.left;
    while (target.kind === NonterminalKind.ParenthesizedExpression) {
      target = target.expression;
    }
    // Need to handle qualified names
    if (target.kind === TerminalKind.Identifier
        || target.kind === NonterminalKind.MemberAccess) {
      // More complex logic...
    }
  }
  // Don't forget to recurse!
  for (const child of node.children) {
    findAssignments(child);
  }
}
```

➔ Recursion management
➔ State tracking

Common Challenges with Manual Traversal

```
function findAssignments(node: CstNode) {
  if (node.kind === NonterminalKind.Assignment) {
    // Need to handle parenthesized expressions
    let target = node.left;
    while (target.kind === NonterminalKind.ParenthesizedExpression) {
      target = target.expression;
    }
    // Need to handle qualified names
    if (target.kind === TerminalKind.Identifier
        || target.kind === NonterminalKind.MemberAccess) {
      // More complex logic...
    }
  }
  // Don't forget to recurse!
  for (const child of node.children) {
    findAssignments(child);
  }
}
```

➔ Recursion management
➔ State tracking
➔ Error handling complexity

Common Challenges with Manual Traversal

```
function findAssignments(node: CstNode) {
  if (node.kind === NonterminalKind.Assignment) {
    // Need to handle parenthesized expressions
    let target = node.left;
    while (target.kind === NonterminalKind.ParenthesizedExpression) {
      target = target.expression;
    }
    // Need to handle qualified names
    if (target.kind === TerminalKind.Identifier
        || target.kind === NonterminalKind.MemberAccess) {
      // More complex logic...
    }
  }
  // Don't forget to recurse!
  for (const child of node.children) {
    findAssignments(child);
  }
}
```

- ➔ Recursion management
- ➔ State tracking
- ➔ Error handling complexity
- ➔ Edge case proliferation

Common Challenges with Manual Traversal

```
function findAssignments(node: CstNode) {
  if (node.kind === NonterminalKind.Assignment) {
    // Need to handle parenthesized expressions
    let target = node.left;
    while (target.kind === NonterminalKind.ParenthesizedExpression) {
      target = target.expression;
    }
    // Need to handle qualified names
    if (target.kind === TerminalKind.Identifier
        || target.kind === NonterminalKind.MemberAccess) {
      // More complex logic...
    }
  }
  // Don't forget to recurse!
  for (const child of node.children) {
    findAssignments(child);
  }
}
```

- ➔ Recursion management
- ➔ State tracking
- ➔ Error handling complexity
- ➔ Edge case proliferation
- ➔ Maintenance burden

Query-Based Approach

```
const query = Query.parse(`
  @vardecl [VariableDeclaration
    [TypeName ["uint"]] [Identifier]
  ]
`);
```

Query-Based Approach

```
const query = Query.parse(`
  @vardecl [VariableDeclaration
    [TypeName ["uint"]] [Identifier]
  ]
`);

// Find immediately nested function declarations
const nestedFuncsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        [FunctionDefinition
          [Block [Statement @nested [FunctionDefinition]]]]
      ]
    ]
  ]
`);
```

Query-Based Approach

```
const query = Query.parse(`
  @vardecl [VariableDeclaration
    [TypeName ["uint"]] [Identifier]
  ]
`);

// Find immediately nested function declarations
const nestedFuncsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        [FunctionDefinition
          [Block [Statement @nested [FunctionDefinition]]]]
      ]
    ]
  ]
`);
```

➔ Declarative syntax patterns

Query-Based Approach

```
const query = Query.parse(`
  @vardecl [VariableDeclaration
    [TypeName ["uint"]] [Identifier]
  ]
`);

// Find immediately nested function declarations
const nestedFuncsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        [FunctionDefinition
          [Block [Statement @nested [FunctionDefinition]]]]
      ]
    ]
  ]
`);
```

- ➔ Declarative syntax patterns
- ➔ Structure-aware queries

Query-Based Approach

```
const query = Query.parse(`
  @vardecl [VariableDeclaration
    [TypeName ["uint"]] [Identifier]
  ]
`);

// Find immediately nested function declarations
const nestedFuncsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        [FunctionDefinition
          [Block [Statement @nested [FunctionDefinition]]]]
      ]
    ]
  ]
`);
```

➔ Declarative syntax patterns
➔ Structure-aware queries
➔ Composable patterns

Advanced Query Patterns

```
// Find function calls immediately inside unchecked blocks
const uncheckedCallsQuery = Query.parse(`
  [UncheckedBlock
    [Statement
      @calls [FunctionCall]
    ]
  ]
`);
```

Advanced Query Patterns

```
// Find function calls immediately inside unchecked blocks
const uncheckedCallsQuery = Query.parse(`
  [UncheckedBlock
    [Statement
      @calls [FunctionCall]
    ]
  ]
`);

// Find state variable declarations with complex types
const complexStateVarsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        @vardecl [VariableDeclaration
          [TypeName (
            [Mapping] |
            [ArrayTypeName]
          )+]
        ]
      ]
    ]
  ]
`);
```

Advanced Query Patterns

```
// Find function calls immediately inside unchecked blocks
const uncheckedCallsQuery = Query.parse(`
  [UncheckedBlock
    [Statement
      @calls [FunctionCall]
    ]
  ]
`);

// Find state variable declarations with complex types
const complexStateVarsQuery = Query.parse(`
  [ContractDefinition
    [ContractMembers
      [ContractMember
        @vardecl [VariableDeclaration
          [TypeName (
            [Mapping] |
            [ArrayTypeName]
          )+]
        ]
      ]
    ]
  ]
`);
```

Structural Analysis Use Cases

Syntax Validation

- ➔ Custom coding standards
- ➔ Project-specific restrictions
- ➔ Version compatibility checks

Structural Analysis Use Cases

Syntax Validation

- ➔ Custom coding standards
- ➔ Project-specific restrictions
- ➔ Version compatibility checks

Style Checking

- ➔ Naming patterns
- ➔ Structure conventions
- ➔ Context-aware rules

Structural Analysis Use Cases

Syntax Validation

- ➔ Custom coding standards
- ➔ Project-specific restrictions
- ➔ Version compatibility checks

Style Checking

- ➔ Naming patterns
- ➔ Structure conventions
- ➔ Context-aware rules

Pattern Detection

- ➔ Anti-patterns
- ➔ Optimization opportunities
- ➔ Complex structural patterns

More Applications

Code Transformation

- ➔ Automated refactoring
- ➔ Code modernization
- ➔ Automated modifications
- ➔ Formatting

More Applications

Code Transformation

- ➔ Automated refactoring
- ➔ Code modernization
- ➔ Automated modifications
- ➔ Formatting

Documentation Generation

- ➔ Function signature extraction
- ➔ Structure analysis
- ➔ Usage pattern documentation
- ➔ Comment processing

Key Benefits for Syntax Analysis

➔ Structure-aware queries

Key Benefits for Syntax Analysis

- ➔ Structure-aware queries
- ➔ Complete syntax preservation

Key Benefits for Syntax Analysis

- ➔ Structure-aware queries
- ➔ Complete syntax preservation
- ➔ Efficient pattern matching

Key Benefits for Syntax Analysis

- ➔ Structure-aware queries
- ➔ Complete syntax preservation
- ➔ Efficient pattern matching
- ➔ Composable syntax rules

Key Benefits for Syntax Analysis

- ➔ Structure-aware queries
- ➔ Complete syntax preservation
- ➔ Efficient pattern matching
- ➔ Composable syntax rules
- ➔ Maintainable analysis code

Impact on Development

Before Query API

- ➔ Complex traversal code
- ➔ High maintenance burden
- ➔ Brittle implementations
- ➔ Difficult to extend

Impact on Development

Before Query API

- ➔ Complex traversal code
- ➔ High maintenance burden
- ➔ Brittle implementations
- ➔ Difficult to extend

With Query API

- ➔ Clear, focused code
- ➔ Easy to maintain
- ➔ Robust implementations
- ➔ Highly extensible

Questions?

Resources

- ➔ **GitHub**: github.com/NomicFoundation/slang
- ➔ **Docs**: nomicfoundation.github.io/slang/latest/
- ➔ **Community**: t.me/+ild4-RjaDqgxNjEx

Take a picture of these links to reference later!

NOMIC
FOUNDATION