Solgrep - A grammar aware Solidity query tool

Ferran Celades Pons
Master's degree in Information and Communication Technologies Security
Protocols and security applications

Tutor: Alberto Ballesteros Rodríguez

31 de Maig de 2022

FINAL WORK SHEET

Title:	*Solgrep - A grammar aware Solidity query tool*
Author's name:	*Ferran Celades Pons*
Tutor name:	*Alberto Ballesteros Rodríguez*
Delivery date (mm/aaaa):	*05/2022*
Degree:	*Master's degree in Information and Communication Technologies Security*
Work Area:	*Protocols and security applications*
Language:	*English*
Keywords:	*Grammar, Search, Queries, AST, Tree, Ellipsis, Nodes, Metavars, Semgrep, YAML, Rules*

Resum del Treball:

En els últims anys, l'ús de la cadena de blocs, o blockchain, ha anat creixent ràpidament. Moltes indústries han estat utilitzant i desenvolupant plataformes de cadena de blocs per realitzar càlculs descentralitzats. Una de les tecnologies subjacents més populars en blockchain es Ethereum.
S'han escrit moltes eines per analitzar i trobar vulnerabilitats als contractes intel · ligents de Solidity des d'un punt de vista estàtic [3] i dinàmic [4]. Tanmateix, cap de les eines ja desenvolupades permet una aportació fàcil per part de la comunitat sense haver de modificar el propi codi font de l'eina o escriure consultes específiques de sintaxi complexes.

En aquest projecte he creat Solgrep. Solgrep és una eina que permet la cerca semàntica estàtica al codi Solidity. La idea inicial de Solgrep s'havia d'utilitzar com a part de Smart Contracts Solidity Audits com una eina remarcable en l'arsenal d'un auditor. Tanmateix, es va observar que aquesta eina es podria integrar fàcilment amb les piles de desenvolupament de Solidity actuals per trobar patrons dolents comuns i errors de codificació que solen fer els desenvolupadors de Solidity.
Aquest projecte constarà de tres parts diferents, més una obra ampliada que s'acabarà parcialment i es deixarà a la comunitat:

La primera fase serà la creació d'una utilitat que sigui capaç d'analitzar qualsevol codi de Solidity en un d'arbre abstracte (AST) que després es pugui interpretar i treballar. Per a aquesta fase, s'utilitzarà la biblioteca analitzadora tree-sitter[5] amb una gramàtica de Solidity personalitzada.

La segona fase consistirà a escriure un programa que sigui capaç de comparar i extreure dos AST proporcionats, l'arbre del codi font i l'arbre proporcionat per l'usuari com a consulta.

En la tercera fase, l'eina s'ampliarà amb un sistema de nidificació que permetria fusionar i restar múltiples consultes entre si. Per exemple, trobar un patró dins d'un patró, trobar un patró que no estigui dins d'un altre patró o trobar un patró que contingui diversos patrons. El sistema de consultes es gestionarà i es configurarà mitjançant fitxers YAML. S'utilitzarà la referència sobre la sintaxi de la regla[6].

Com a extensió, l'eina s'utilitzarà per escriure els problemes més importants del registre SWC[7] per mostrar el poder i la simplicitat de l'eina.

Abstract:

In the last few years, the blockchain usage has been growing rapidly. A lot of industries have been using and developing blockchain platforms to perform decentralized computations. One of the most popular underlying technologies is the Ethereum blockchain. Executing, verifying and enforcing credible computable transactions on blockchains is done using smart contracts which the code is written using a Turing complete language [1]. Those contracts are typically written in a high-level programming language called Solidity [2], then compiled to Ethereum Virtual Machine (EVM) assembly instructions and deployed to the Ethereum blockchain.

So many tools have been written to analyze and find vulnerabilities in Solidity smart contracts from a static [3] and dynamic standpoint [4]. However, none of the already developed tools allow easy contribution by the community without actually having to modify the tool source code itself or write complex syntax specific queries.

In this project we have created Solgrep. Solgrep is a tool that allows static semantic aware search on Solidity code. The initial idea of Solgrep was to be used as part of Smart Contracts Solidity Audits as a remarkable tool in the arsenal of an auditor. However, it was noticed that this tool could be easily integrated with current Solidity development stacks to find common bad patterns and coding mistakes that Solidity developers tend to do.

Contents

List of Figures

1 Introduction

1.1 State of Art

In the last few years, the blockchain usage has been growing rapidly. A lot of industries have been using and developing blockchain platforms to perform decentralized computations. One of the most popular underlying technologies is the Ethereum blockchain. Executing, verifying and enforcing credible computable transactions on blockchains is done using smart contracts which the code is written using a Turing complete language [1]. Those contracts are typically written in a high-level programming language called Solidity [2], then compiled to Ethereum Virtual Machine (EVM) assembly instructions and deployed to the Ethereum blockchain.

So many tools have been written to analyze and find vulnerabilities in Solidity smart contracts from a static [3] and dynamic standpoint [4]. Static analysis is one of the most effective ways to detect potential issues in contracts. Usually, static analysis tools work by analyzing the source code or a disassembled version of it and then transforming it into an internal representation where the actual analysis and detection is performed.

None of the already developed tools allow easy contribution by the community without actually having to modify the tool source code itself or write complex syntax specific queries. Some tools that allow static code querying do exist such as semgrep. Semgrep [8] was written to find bugs in source code with extensibility and usability in mind and without having to actually rewrite the source code of the utility.

However, customization and extensibility to detect flow aware bugs such as Solidity Re-entrance bugs are hard to be detected without execution trace or further analysis.

1.2 Solution

In this project we have created Solgrep. Solgrep is a tool that allows static semantic aware search on Solidity code. The initial idea of Solgrep was to be used as part of Smart Contracts Solidity Audits as a remarkable tool in the arsenal of an auditor. However, it was noticed that this tool could be easily integrated with current Solidity development stacks to find common bad patterns and coding mistakes that Solidity developers tend to do.

The search queries are written using plan Solidity and then parsed into an intermediate AST for interpretation. The grammar used to parse the Solidity files and user provided queries was extended with syntax support for ellipsis (aka skipping sibling/child nodes on the AST). The query tree and the source code tree are then compared using a BFS algorithm to determine if both trees are equal or not. Node comparison do include regex support and metavars definitions for matching the most complex rules that you could possibly imagine. The system allows writing the rules in a YAML file which

will be used to query the source code. It allows the community to write rules in an easy YAML syntax which can be re-used an extended as wanted.

1.3 Objectives

This project will consist in three different parts, plus an extended work that will be partially completed and left to the community:

The first phase will be creating an utility that is able to parse any Solidity code in an Abstract Tree object (AST) that can later be interpreted and worked on. For this phase, the tree-sitter[5] parser library with a custom made Solidity grammar will be used. The grammar will be modified and updated with the latest Solidity features until a 100% code coverage (or close) is achieved on public Solidity projects.

The second phase will consist in writing a program that is able to compare and extract two provided AST's, the source code tree and the user provided tree. The AST comparing code will check for the tree node contents and compare them based on predefined rules. The provided user AST will allow skipping nodes by using an ellipsis syntax, similar to a glob or regex asterisk, which will allow skipping depth nodes and sibling nodes. During this phase, the Solidity grammar will be extended to support ellipsis on all component. Reference on pattern syntax[9] will be used.

In the third phase the tool will be extended with a nesting system that would allow multiple queries to be merged and subtracted from each other. As an example, finding a pattern inside a pattern, finding a pattern that is not inside of another pattern or finding a pattern that contains multiple patterns. For this, conditional queries will be implemented based on previous explored AST results. For all the found queries, the system will generate a report, using a custom provided user message, the line and content of all the found queries. The query system will be managed and configured using YAML files. Reference on rule syntax[6] will be used.

As an extension, the tool will be used to write the most important SWC Registry[7] issues to showcase the power and simplicity of the tool.

1.4 Methodology

During phase one, the code will be base on an already made tree-sitter parser for Solidity [10]. However, code modifications will be needed since some features are not supported on the parser. The top 30 projects on Github with Solidity code will be used for code coverage tests.

For phase two, the utility will be written using Python since the tree-sitter binding for this programming language exists. The code will be extensively tested and simplified versions of the AST will be used before testing with the real Solidity AST. The AST ellipsis system will be tested using simplified and handwritten ASTs.

For phase three, the querying system will be storing each of the found results. Those results will be used in combination with a custom provided user message to have a final report. The template system will use jinja2 [11] for the generation.

On the extension phase, each issue demo code present on the SWC Registry will be taken and a general rule for it will be written and tested.

1.5 Tasks

- State of the art: This task will consist on searching for existing tools to solve the problem and see how we can re-use part of it for our project idea.

- Design: During this task we will design our tool and the expected behavior for it. We will be extending the grammar syntax for Solidity to support our query syntax and write the underlying functionality of the tool.

  - Extend Solidity and write an AST query program (Initial AST ellipsis support). (PEC 2 & PEC 3)
  - Support ellipsis on Solidity grammar and different types node comparison for Solidity. (second half of PEC 3 & first half of PEC 4)
  - Write the template system and complex nested queries (Test rules with SWC Registry). (PEC 4)

- Evaluation: This task consist on using our tool ans seeing how good it is and what could be improved by using it against real world scenarios.

  - Provide the SWC Registry queries using the created tool. (PEC 4)

The tasks are represented on the Figure 1 Gantt chart. These dates are approximate, and a margin has been given for each activity in case of setbacks. This chart does not include the times for the video delivery task, which it is estimated to by less than 1 week.

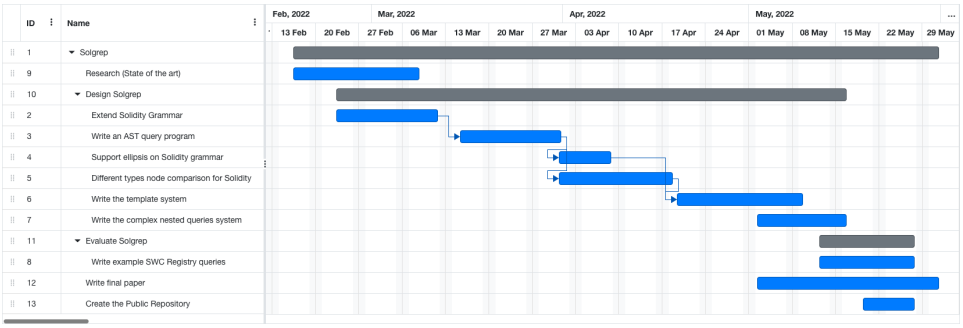

Figure 1: Gantt chart for all the tasks and the programmed time for them

2 Grammar

A programming language grammar is a set of instructions, given in the form of rules, about how to write statements that are valid for that programming language. Those rules specify how characters and words can be put one after the other to form a valid statement on that language. The rules defining how words and characters are put together are called lexing rules, and are defined under a lexer. Once the lexer stops processing certain rule it will generate a "token". A "token" is a valid lexical group. Later, those tokens are interpreted by other rules called parsing rules. The parsing rules, defined under a Parser, define the relationship between tokens valid for that specific programming language.

Lets use an example snippet, Listing 1, that could be used for almost any programming language .

Listing 1: Programming language code example

```
1 1234 + 456
```

The program performing lexical token extraction and parsing is usually called a "grammars" or "grammar parsers", taking the name from the last step of the flow being actually the Parser itself. Figure 2 is displaying the flow that a grammar parser will perform. A grammar lets us transform a program, which is normally represented as a linear sequence of ASCII characters, into a syntax tree.

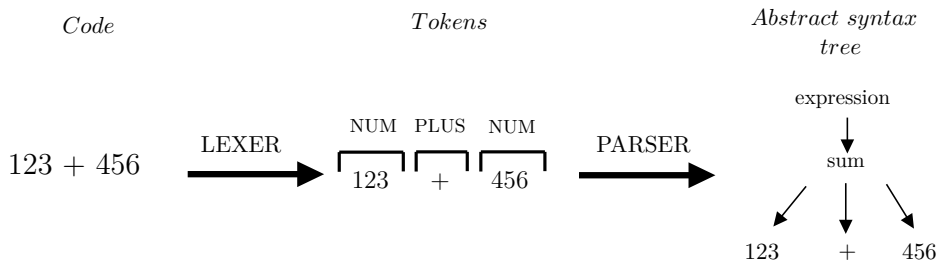

Figure 2: Flow for a grammar

The output of a parser is usually a custom formatted Abstract Syntax Tree (AST). Usually parsers provide a way to have the AST represented in many different ways and provide programming language bindings to walk and traverse the trees.

2.1 Describing grammars

There exist multiple languages themselves that are written and used to describe the grammar for other languages, such an example is ANTLR [12].

ANTRL, is used to write aggregated rules in a key-value form using specific syntax that can describe an entire programming language. As an example, Solidity does provide an official description for the Lexer [13] and Parser [14].

2.1.1 Tree-sitter

Tree-sitter is a parser generator tool that compiles a grammar description into a POSIX library. The library does expose some methods that parsing any file input and walking though the parsed AST. It does present with many language bindings [15] that allow direct interaction with the library methods by exposing APIs. There is currently one implementation of the Solidity grammar using tree-sitter syntax [10]. This project will extend and update the aforementioned grammar with the latest updates [16] under Github [17].

3 Tree-sitter changes and updates

This section will explain all changes that have been made on the original tree-sitter developed by Joran Honig [10].

The grammar file does cover almost the entire syntax for the latest version of Solidity, currently siting under 0.8.9. However, old compatibility features were missing including but not limited to:

- Experimental Pragma support, such as `SMTChecker` and `ABIEncoderV2` and `abicoder v2` on `solidity 0.7.6`
- Missing support for tuple variable declaration, such as `(address x, address y)`
- Declaring function and event parameters as function types was not allowed.
- Usage of `var` keyword to declare variables and tuple variables such as `(c,d)= (1,2)`.
- Missing some reversed keywords such as `after`.
- Missing test cases for structure inline initialization such as `MapEntry({ _key: value, _value: 2 })`
- Missing partial support for underscore numbers and hex digits, such as `1_123`
- Yul expressions were missing `yul_evm_builtin` support for solidity `< 0.6.0`.
- Slicing access members were not declared as optional, slices like the following `[4:]` were not supported.
- Function type did not support neither, parameters, visibility or mutability.

Some changes on newer versions were introduced and were not reflected on the grammar, some of them are:

- The fallback functions do not return values on `>=0.6.0`, this was added on the `fallback_receive_definition` description.
- A new block type was added on `>0.8.0` named `unchecked`. The support was added as well.

Both compatibility and additions were merged into a single commit hash `1b2ade71d54b0edaa1e932d9700de811568b932e`[17]. This commit includes the previous compatibility fixed issues and the newly added code to support the latest solidity compilers. Furthermore, the test cases and corpus samples were updated to reflect the changes and test compatibility with existing code. The repository containing the changes can be found publicly under Github[17].

3.1 Testing coverage

Once the test cases were extended an verified, as shown in Figure 3, it was time to test our newly added features and fixed issues with real world code. For that, the `most-starred-for-languages` script[17] was used to obtain the TOP 30 projects written in solidity present on Github.

Figure 3: All written testcases for the tree-sitter solidity repository

Listing 2: TOP 30 projects on Github containing solidity code

```
https://github.com/ethereum/EIPs
https://github.com/Aircoin-official/AirCash
https://github.com/Dapp-Learning-DAO/Dapp-Learning
https://github.com/fravoll/solidity-patterns
https://github.com/willitscale/learning-solidity
https://github.com/sushiswap/sushiswap
https://github.com/Rari-Capital/solmate
https://github.com/crytic/not-so-smart-contracts
https://github.com/compound-finance/compound-protocol
https://github.com/crytic/echidna
https://github.com/nibbstack/erc721
https://github.com/ExtropyIO/defi-bot
https://github.com/xtblock/xtt
https://github.com/solidlyexchange/solidly
https://github.com/Arachnid/solidity-stringutils
https://github.com/studydefi/money-legos
https://github.com/PatrickAlphaC/nft-mix
https://github.com/crytic/building-secure-contracts
https://github.com/provable-things/ethereum-api
https://github.com/Uniswap/v2-periphery
https://github.com/OlympusDAO/olympus-contracts
https://github.com/safemoonprotocol/Safemoon.sol
https://github.com/yam-finance/yam-protocol
https://github.com/zeriontech/defi-sdk
https://github.com/unlock-protocol/unlock
https://github.com/andrecronje/rarity
https://github.com/aragon/aragonOS
https://github.com/HashLips/hashlips_nft_contract
https://github.com/makerdao/multicall
```

After the list was complete, the command shown under Listing 3 was executed to clone the repositories under the sample folder.

Listing 3: Cloning top 30 solidity projects under the test/sample folder

```
cat projects.txt | xargs -L 1 -I {} git -C ./test/sample clone
    --depth=1 {}
```

After the repositories were cloned the `tree-sitter` utility with the stats command was as shown in Listing 4 was executed.

Listing 4: Command executed to stat the parsing ratio on the sample repos

```
$ tree-sitter generate
$ tree-sitter parse 'test/sample/**/*.sol' --stat --quiet
...
Total parses: 1479; successful parses: 1453; failed parses: 26;
     success percentage: 98.24%
```

Out of `1479` files, `1453` were successfully parsed and only `26` had some parsing issues. However, those issues do not mean that the entire file was

not parsable but the stat tool reported it as invalid. Thats a full parsing success ratio of **98.24%** and a coverage of over **99.7%**. The none-parsable issues were related to bad code implementations and old solidity versions structure manipulations, futhermore some yul `loop` old behaviors were also not parsed.

> This coverage was enough to continue and assure that almost all tokens will be parsed and interpreted into a valid AST. If any none valid node is found, direct regex comparison will be made.

4 Understanding the AST

For this section the Listing 5 code will be used. It contains a simple external function written in Solidity that performs no action at all.

Listing 5: Sample Solidity code

```
pragma solidity ^0.8.4;


// Comment
contract TestContract {

  function test() external {
  }
}
```

The Listing 5 code is interpreted by tree-sitter and parsed as shown in Listing 7. This is an AST representation where each level is added under a parenthesis (()). The AST does show the starting and ending line and character inside that line.

Listing 6: Output generated by tree-sitter when parsing the sample code

```
(source_file [0, 0] - [8, 1]
  (pragma_directive [0, 0] - [0, 23]
    (solidity_directive [0, 7] - [0, 22]
      (pragma_versions [0, 16] - [0, 22])))
  (comment [3, 0] - [3, 11])
  (contract_declaration [4, 0] - [8, 1]
    name: (identifier [4, 9] - [4, 21])
    body: (contract_body [4, 22] - [8, 1]
      (function_definition [6, 2] - [7, 3]
        function_name: (identifier [6, 11] - [6, 15])
        (parameter_list [6, 15] - [6, 17])
        (visibility [6, 18] - [6, 26])
        body: (function_body [6, 27] - [7, 3])))))
```

By using a traverser (Listing 7), the Python binding library[18] and the AnyTree library [19] we can obtain a representation of this tree as show in Figure 4

Listing 7: Output generated by tree-sitter when parsing the sample code

```
class TreeNode(NodeMixin):
  def __init__(self, type, node, parent=None, children=None):
      super(TreeNode, self).__init__()
      self.name = "{}".format(type)
      self.type = type
      self.node = node
      self.parent = parent

```

```
\pagebreak
# root = Treesitter root

def _parse_node(_node, _last_parent):
    node = TreeNode(_node.type, _node, parent=_last_parent)

def _traverse(self):
    last_parent = None
    def _traverse(node, last_parent):
        last_parent = self._parse_node(node, last_parent)
        for child in node.children:
            _traverse(child, last_parent)
    _traverse(root, last_parent)
```

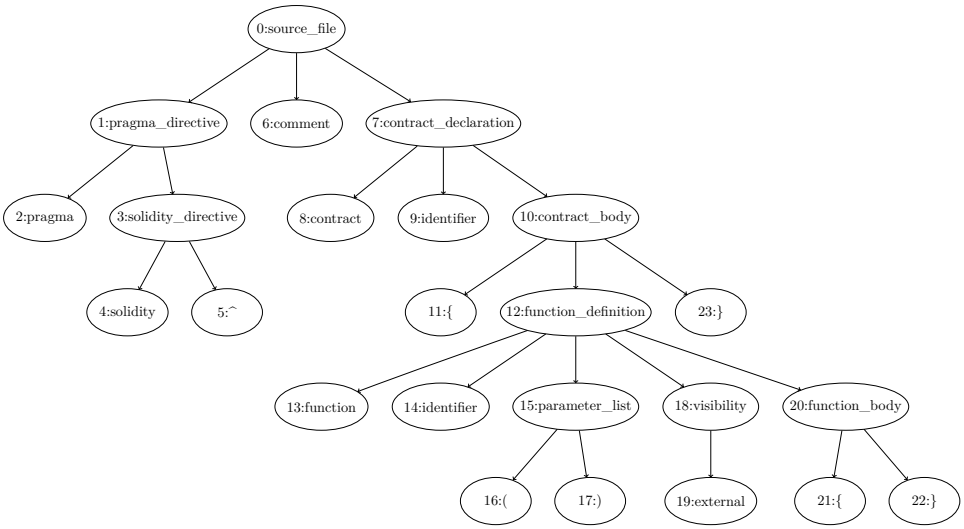

Figure 4: AST representation of the sample code

As we can see in Figure 4, the AST level is determined by the scope depth. If the code does not contain any sub-statement the tree heigh will remain on the same level.

5 Implementing a tree comparer

As seen in Section 4, the tree depth grows in relation with the sub-statements of the source code.

For simplicity, lets take the source code from Listing 8 which can be represented with the AST shown in Figure 5. As seen on the source code the code does contain a `test()` function inside the contract scope. This corresponds to the level of nodes `6` to `14` on the AST representation.

Listing 8: Sample Solidity code

```
1 contract TestContract {
2   function test() {
3
4   }
5 }
```

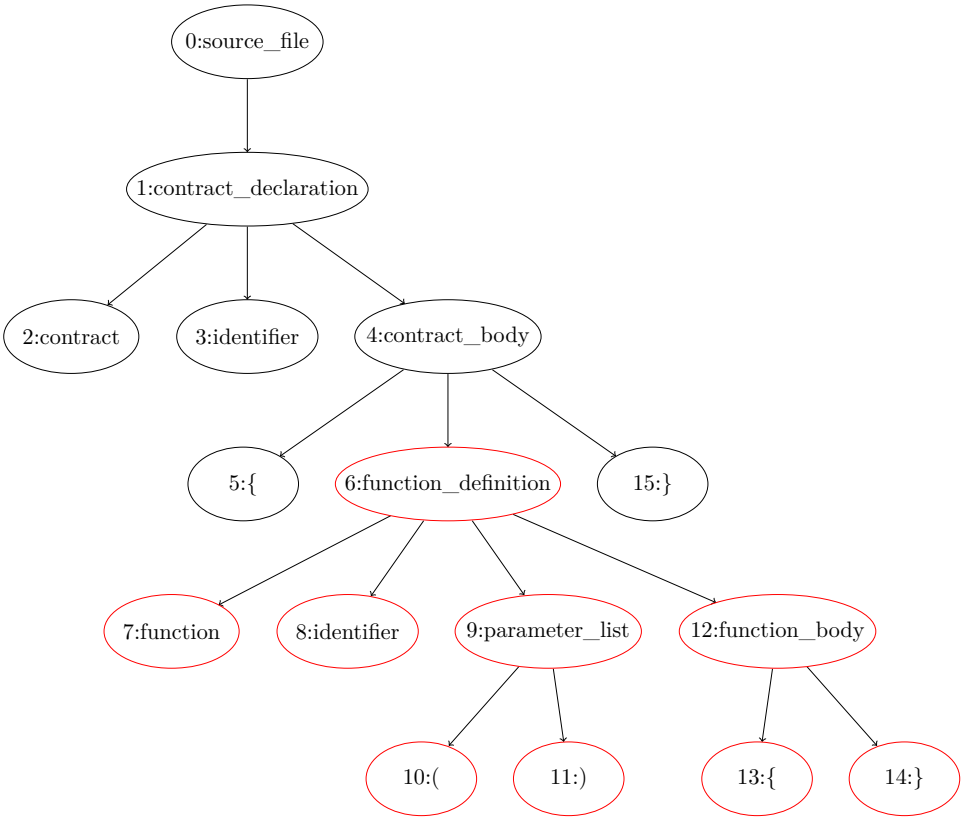

Figure 5: AST representation of the sample code

Imagine that we want now to match a `Contract` declaration with any function on it. Having a tree that matches this statement would require

knowing all possible definitions of the contract. However, it is possible to create a new node type which represents any sub-level nodes or any sibling nodes. This new type is called “ellipsis”.

5.1 Ellipsis node

The ellipsis node, is represented with ..., this statement matches any subnode and sibling nodes until a matching none-ellipsis node is found or until the tree level is exhausted.

Keeping the Listing 8 example, we could match any contract declaration with the statement shown in Listing 9

Listing 9: Sample Solidity code with ellipsis

```
1 contract TestContract {
2   ...
3 }
```

The AST representation in conjunction with the old AST, can be seen in Figure 6. The ellipsis node, will match any sub-statement including sibling nodes. For example, if the original source code had multiple function definitions, the node level `5:` will contain two `function_definition` nodes. The ellipsis, will still match both of those definitions until a none-ellipsis node is found or the level is exhausted.

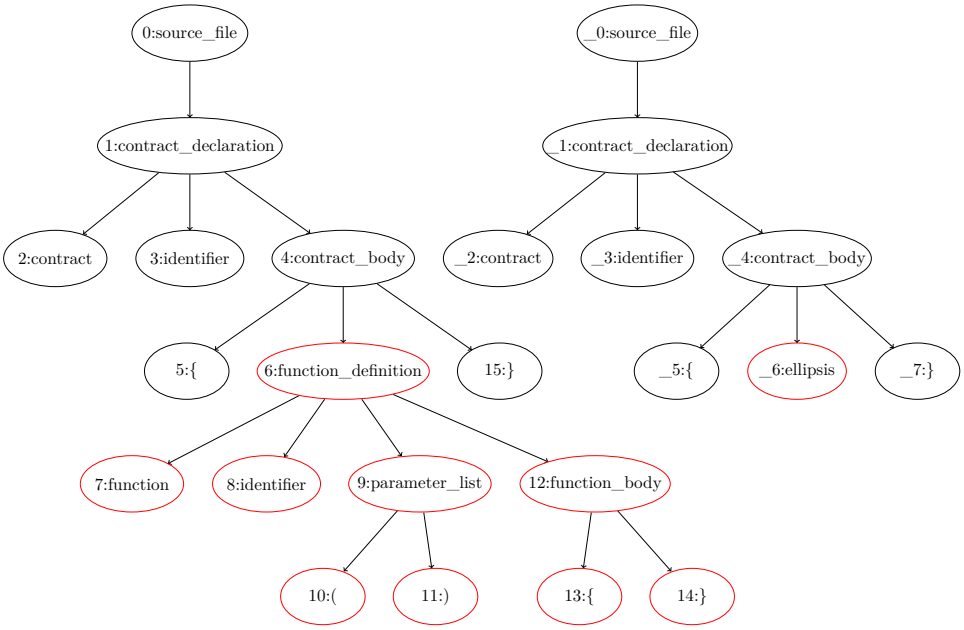

Figure 6: AST representation of the sample code with the ellipsis definition

5.2 Comparing Trees

The search will be performed using a BFS algorithm [20] to search for the first node that do match the query root node. Once the root query node is found on the source code, the query tree will be traversed using BFS until the query tree is exhausted. Once the traversal does complete on the query tree, and all depths were equal, we can assure that a result for the query on the original source was found. Each depth will be compared using the depth level comparer algorithm that implement ellipsis and node skipping comparison.

Taking as a reference the Figure 6 on the 5th node level, the sequence could be represented as ilustrated on Listing 10. A depth level comparer (for the BFS [20] algorithm) was written to ilustrate this concept, the code can be found under the Solgrep repo examples/comparer.py[21]. The logic of the code is to represent the tree level as an array and match it against other arrays containing ellipsis node (represented with a single dot . for simplicity). Some test scenarios of matching levels can be seen on Listing 11.

Listing 10: 5th level of the ellipsis demo represented using an array

```
original = ['{', 'function_definition', '}']
comparer = ['{', '.', '}']
```

Listing 11: Sample Solidity code with ellipsis

```
a = [1, 2, 3, 4, 5]
b = ['.', 2, '.', 5]
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
b = ['.', 5]
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
b = [1, '.', 5]
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
b = [1, 2, '.']
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
b = ['.']
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
b = [1, '.', '.', '.', 4, '.', '.', 5]
assert compare(a, b) == True

a = [1, 2, 3, 4, 5]
```

```
b = [1, '.', '.', 4]
assert compare(a, b) == False

a = [1, 2, 3, 4, 5]
b = ['.', '.', 4]
assert compare(a, b) == False
```

5.3 Extending tree-sitter-grammar with ellipsis support

The updated tree-sitter solidity grammar [17], should be extended to support ellipsis syntax on the desidered nodes. For it, the `tree-sitter` nodejs library, allows extending from an original grammar with syntax shown in Listing 13. The grammar definition can now extend the original grammar, and replace or add new parsable tokens on any declaration under the `rules`.

Listing 12: Original grammar extension template

```
const standard_grammar = require('./tree-sitter-solidity/
    grammar.js');
module.exports = grammar(standard_grammar, {
  name: 'solgrep',

  rules: {

  }
});
```

As an example, adding support for the ellipsis token on any standard statement and inside a contract body can be done by defining the rules shown in Listing 13 under the `rules` of the extended grammar.

Listing 13: Rules added to the extended grammar to support basic ellipsis

```
_contract_body: ($, previous) => {
  return choice(
    ...previous.members,
    $.ellipsis
  );
},

_expression: ($, previous) => {
  return choice(
    ...previous.members,
    $.ellipsis,
  );
},

ellipsis: $ => '...',
```

Compiling the rules from Listing 13 and executing the `tree-sitter` parse utility against the Listing 9 code does produce the `tree-sitter` representation shown on Listing 14

Listing 14: Result of extending the original grammar with ellipsis support under contracts

```
(source_file [0, 0] - [2, 1]
  (contract_declaration [0, 0] - [2, 1]
    name: (identifier [0, 9] - [0, 21])
    body: (contract_body [0, 22] - [2, 1]
      (ellipsis [1, 2] - [1, 5]))))
```

5.3.1 Comparing Nodes and metavar support

By definition, two nodes will be the same if the underlying tokens do match on every single element. As an example, the contract name `ContractName` will only match with compared node if every single character is the same, in this case the comparer node should be `ContractName`.

For example, writing a query for the code shown on Listing 15 that would match both, `func1` and `func2`, the latter calling the former, could be possible accomplished by using ellipsis as shown on the Listing 16. However, that would also match `func3` being called by `func4` and moreover, `func4` calling `func1`, which is not something that the code states. Since ellipsis can match anything it is possible to match invalid statements that do not truly represent the real source code.

Listing 15: Example code showing call dependencies

```
contract ContractName {
  function func1() external {}
  function func2() external {
    func1();
  }

  function func3() external {
  }
  function func4() external {
    func3();
  }
}
```

Listing 16: Query to match the example code with call dependencies using ellipsis

```
contract ContractName {
  ... // match any previous code
  function ...() external {}
```

```
4   ...
5   function ...() external {
6     ...();
7   }
8   ... // match any following code
9 }
```

With the implications seen on relying only on ellipsis a new way of representing tokens was introduced. Any identifier under solgrep can be prefixed with the `$` symbol (dollar sign). When comparing literal tokens, aka nodes, this value will be checked. If the identifier starts with this symbol, solgrep will keep a reference to its literal value which can be later referenced during the query. As an example, Listing 16 could be rewritten as shown in Listing 17.

Listing 17: Query to match the example code with call dependencies using metavars

```
1 contract ContractName {
2   ... // match any previous code
3   function $FNC() external {
4     ...;
5   }
6   ...
7   function $CALLER() external {
8     $FNC();
9   }
10  ... // match any following code
11 }
```

The code will now keep valid metavars references and find all possibilities that do match those metavars. Solgrep, will keep a reference of valid metavars while scanning the query tree. With the Listing 17 query, the `$FNC` metavar will initially be filled by `func1`, `func2`, `func3` and `func4`, since they all match the `$FNC` definition (there is an internal ellipsis indicating that this function can contain "any" or "none" body). Once solgrep does start to interpret the `$CALLER` definition, the `func2` and `func4` literals on `$FNC` will be discarded. The `$CALLER` metavar will be store with `func2` and `func4` as valid.

> The metavar system will never detect false positives since the literal representation of the placeholder metavar variables is compared against the queried source code.

Some internal metavars are also defined, for example, `$TYPE a = 0;` query could be used to match `bool a = 0;`.:

- `$TYPE`: It will match any type, such as `uint256`, `bool`, `bytes`.
- `$VISIBILITY`: It will match any function visibility, such as `public`, `external`, `internal`.

- `$STATE`: It will match any function mutability, such as `view`, `pure`.
- `$STORAGE`: It will match any type memory storage, such as `memory`, `storage`, `calldata`.
- `$VERSION`: It will match any pragma solidity version, such as `0.8.4`, `>=7.0.0`.
- `$EXPERIMENTAL`: It will match any pragma experimental string, such as `ABIEncoderV2`, `SMTChecker`.

Once a metavar is used, the literal value that reference to is keep on the placeholder. This means, that further references to the same metavar do hold the last value. Sometimes, we do want to use those internal vars to match complex conditions, as seen in Listing 18. The previous stated listing, does match any function with any name, that takes one parameter of any type and returns a value of the same type. This function should call any function that takes the passed argument and the returned call value should be stored on a variable and returned from the main function.

If we now want to have the same query but with the possibility of the returned value being the same or different type as the parameter type that would not work. The parameter type and return type should be the same as defined on the query.

Thats why, internal metavars do support enumeration by appending a number to the definition. And as previous metavars identifiers they will hold the first value and all possible values that it matches. As an example, Listing 18 could be rewritten to support different argument and return values as shown in Listing 19, which defines two different any type, `$TYPE0` and `$TYPE1`.

Listing 18: Complex internal metavar dependencies

```
1 contract $CONTRACT {
2
3   function $FNC1($TYPE $VAR1) $VISIBILITY returns($TYPE){
4     ...
5     $TYPE $VAR2 = $FNC2($VAR1);
6     ...
7     return $VAR2;
8   }
9 }
```

Listing 19: Complex internal metavar dependencies with metavar enumeration

```
1 contract $CONTRACT {
2
3   function $FNC1($TYPE0 $VAR1) $VISIBILITY returns($TYPE1){
4     ...
5     $TYPE1 $VAR2 = $FNC2($VAR1);
```

```
6       ...
7       return $VAR2;
8     }
9   }
```

6 Solgrep rules

Rules under solgrep are written using `YAML` syntax [22]. The Listing 20 does display an example rule file containing all the components that are required to satisfy a valid `solgrep` rule file.

Listing 20: Example YAML rule file for the solgrep tool

```
1  id: issue-id
2  message: |
3    This is the message {{CONTRACTS | comma}}
4  risk: 1
5  impact: 1
6  patterns:
7    - pattern: contract $CONTRACT {...}
8      and:
9        - pattern: function $FUN(...) ... {...}
10         and:
11         - pattern: ... -= ...
12         - pattern: ... *= ...
13         - pattern: ... += ...
14 metavars-regex:
15   $CONTRACT: .*
16   $FUN: admin.*
```

Each component of the Listing 20 is explained here:

- **id** The id is used to identify the rule and used on the reported in case of multiple rules defined.
- **message** The message is used on the reporter to describe the issue. Placeholders can be used to represent the found metavars, as an example the `{{ CONTRACTS | comma }}` will print in a comma separated list, all the `CONTRACT` metavars that do match the `patterns` section. More on the placeholder under Section 6.1.
- **risk and impact** This is used to represent the severity of the described found issue or rule. Any number or value can be inserted here and will be shown on the reporter.
- **patterns** This is the most complex and were all the solgrep power comes in. This section does allow multiple rules to be concatenated and matched against each other in an hierarchy way. A rule can be searched inside a rule by indenting it and using a combiner node, such as `not`, `and`, `and-either`, `not-either` and more. If multiple patterns are allowed, they can be listed inside `- patterns:`. More on the `patterns` under Section 6.2.
- **metavars-regex** This node does allow describing how metavars will be matched on the system. Since metavars do match a full token

regex patterns are supported. Each token matching the node type for the metavar will be compared with this description, if it does match it will be considered a valid token node. If the type of the node is the same but the regex does not satisfy the token will be considered as different. As an example, Listing 20 does describe 2 metavars, the `CONTRACT` and `FUN`, used on the patterns section to match any contract witch contain any function with a list of patterns inside it. The `CONTRACT` metavars does match any token name `.*`. However, the `FUN` metavar will only match functions starting with `admin` followed by anything, on the function name only. The `*` (asterisk) and `+` (plus) sign on regex will only match the current token, the `*/+` does not match until the end of the line as regex would do.

6.1 Solgrep message placeholders

The message field under the rule file of solgrep does support complex parametrization and placeholders. The system is using jinja2 [11] for the template system. That means that any placeholder as long as it has been defined can be used.

For this section, the Listing 21 contract will be used to showcase all possible scenarios.

Listing 21: Demo code used for placeholder showcasing

```
1 pragma solidity 0.8.12;
2
3 contract Test() {
4
5     uint256 public value;
6
7     function func_add(uint256 a, uint256 b) external {
8         value = a + b;
9     }
10
11    function func_sub(uint256 a, uint256 b) external {
12        value = a - b;
13    }
14
15    function do_multiply(uint256 a, uint256 b) external {
16        value = a * b;
17    }
18 }
```

For every single defined metavar a pluralized placeholder will be used containing a list of all valid tokens for that metavars in case of multiple matches. If a none or single match is found, the pluralized token is still used. As an example, the simplified rule file shown in Listing 22. This rule

file will report the message shown under Listing 23.

Listing 22: Simple rule set showcasing the metavar and message placeholder usage

```
message: The found functions starting with func_ are: {{FUNCS}}
patterns:
  - pattern: function $FUNC(...) ... {...}
metavars-regex:
  $FUNC: func_.*
```

Listing 23: Result for the simple rule set showcasing the metavar and message placeholder usage

```
The found function starting with func_ are: ['func_add', '
    func_sub']
```

The placeholders can be extended by using filters. Filters are a way of manipulating the placeholder information to be represented in a different way. As an example, Listing 24 does display a message rule field and the output for it using the same rule set as Listing 22.

Listing 24: Example of placeholder filtering

```
message: The found function starting with func_ are: {{FUNCS |
    comma}}

Result:

The found functions starting with func_ are: func_add, func_sub
```

The output is filters and comma separated when using the `comma` filter on the placeholder. The system does support multiple placeholders and they can be designed to achieve any output needs:

- **pluralize(list, singular=““, plural=”s”)** The pluralize can be used in combination with a word to pluralize it in case the filter value contains more than 1 item. As an example, `The function{{ FUNCS | pluralize }}` will either return `The function` or `The functions` depending if `FUNCS` has more than one item. The command can be customized on the fly with the arguments. As an example, using is with `There {{ FUNCS | pluralize("is a function", "are multiple functions")}}` would either produce, `There is a function` or `There are multiple functions` depending on the length of the `FUNCS` metavar list.
- **comma(list, wrap=““)** It allows to comma separate the values on a metavar result list and represent them as a string. `{{ FUNCS | comma }}` would produce a string list with all the values separated

by a , character. The comma does accept the `wrap` parameter, which allows adding a token or string before and after each element of the list. As an example `{{ FUNCS | comma('*')}}`, would add ' to the start and end of each element of the metavar list. Listing 22 would produce `The found functions starting with func_ are *func_add*, *func_sub*.`

- **list(list, pattern="{}", endline="\n")** This one allows full customization on the output of the metavar list, by default it does print each element in a new line. For example, this method could be used to create a Markdown[23] list of all elements by using the `{{ FUNCS | list("- {}")}}` filter. You can create the same effect as the `comma` filter by using `{{ FUNCS | list(endline=",")}}`.

Finally, there is one internal placeholder named `CONTENTS`. This placeholder contains a list of the found query results content. As an example, Listing 25 shows the content of this placeholder for the Listing 21 source code and Listing 22 rule file.

Listing 25: Content of the 'CONTENTS' placeholder for the showcase example'

```
['''
    function func_add(uint256 a, uint256 b) external {
      value = a + b;
    }
  ''',
  '''
    function func_sub(uint256 a, uint256 b) external {
      value = a - b;
    }
  ''']
```

6.2 Solgrep patterns

The pattern system allows complex declarations to be formed. It allow multiple rules to be concatenated and matched against each other in an hierarchy way. There exist multiple rules and definitions that can be used with others, and all of them do support metavar expressions inside the rules. There are two categories of rules:

- **Simple rules** They are used to define the query content and used in conjunction with merging rules to create complex queries. They are, `pattern` and `pattern-root`.

- **Merging rules** They are rules which do not contain a query definition by themselves but do use simple rules to create complex

query definitions. They can be combined and merged to satisfy the needs for the query. They are `patterns`, `and`, `not`, `and-either` and `not-either`.

Each rule description and an example showcasing the usage can be found can be found on the following section.

6.2.1 Solgrep patterns rules

- **pattern** This is the simplest definition. It is used to declare a valid `solgrep` rule containing pattern syntax code. It can be used as the base for the `YAML` rule file instead of the `patterns` to only match a single pattern. It can be used in combination with the root `patterns` to match multiple patterns in a single query rule. An example can be seen under Listing 26

Listing 26: Example for the pattern rule

```
1 ...
2 message: ...
3 pattern: function $FUNC(...) ... {...}
4 metavars-regex:
5   $FUNC: .*
6   ...
```

- **patterns** This is used only on the root `YAML` file to indicate that the query does contain complex patterns or combination of different patterns not just a single pattern, although a single pattern can be used as well. An example can be seen under Listing 27 were a query would match any function that starts with either `admin_` or `user_`. This query could be simplified by using a single `- pattern` declaration with an `|` (or) regex expression such as `admin_.*|user_.*`.

Listing 27: Example for the patterns rule

```
1 ...
2 message: ...
3 patterns:
4   - pattern: function $FUNC1(...) ... {...}
5   - pattern: function $FUNC2(...) ... {...}
6 metavars-regex:
7   $FUNC1: admin_.*
8   $FUNC2: user_.*
9   ...
```

- **and** This declaration can only be used in conjunction with a previous `pattern` to match sub-patterns or filter the main pattern for sub-conditions. The Listing 28 example does show a query that would match any function (see the metavar regex expression) and that contains at least one `+` operation with any two operands.

Listing 28: Example for the and rule

```
1 ...
2 message: ...
3 patterns:
4   - pattern: function $VAR(...) ... {...}
5   - and: ... + ...
6 metavars-regex:
7   $VAR: .*
8   ...
```

- **pattern-root** This allows to look for pattern starting from the top of the source file. It can be used in conjunction with the `and` rule to filter other rules based on outer scope patterns. As an example Listing 29 does show the usage of the `root` pattern in combination with the `and` pattern. The example, does find all `-=` and `+=` operations with a top level rule of the pragma version being less than `0.8.0`. This query could also be achieved by filtering the `-=` and `+=` rules with an `and` expression of the pragma. However, that would require writing the same filtering pattern twice.

Listing 29: Example for the pattern-root rule

```
1 ...
2 message: ...
3 patterns:
4   - pattern: ... -= ...
5   - pattern: ... += ...
6   - and:
7     - pattern-root: pragma solidity $VERSION
8 metavars-regex:
9   $VERSION: (\d\.[0-7]\.\d*|<0\.8\.0)
10  ...
```

- **not** This definition is used to filter simple patterns or the results of previous complex patterns for none matching queries. It can be used to filter exceptions for `and` complex rules that would be otherwise complex to achieve with a single or regex expression. An example can be seen in Listing 30. This example, does show a

rule that would match any function but will filter the results with the ones that do not have a visibility set (`VISIBILITY` is an especial metavar type, see Section 5.3.1).

Listing 30: Example for the not rule

```
message: ...
patterns:
  - pattern: |
      function $NAME(...) ... {
        ...
      }
  - not: |
      function $NAME(...) $VISIBILITY {
        ...
      }
```

- **and-either and not-either** At the time of writing this paper, those rules are currently being refactored. They allow as the name states, list multiple simple patterns underneath to obtain multiple `and` results or filter based on multiple `not` rules.

7 SWC

To showcase the power of Solgrep an how it could help on finding already known bugs on Solidity a rule for some of the SWC registry [7] entries were written. Each entry do contain a description of the issue, an example source code and a fixed source code. All the rules can be found under the `/SWC` directory of the main Solgrep repository [21].

As an example, on Listing 31 we can see a Solgrep rule that would match the SWC-100 issue with a message description for it.

Listing 31: Solgrep to find issues for the SWC-100 registry entry

```
1  id: swc-100
2  message: |
3    Functions that do not have a function visibility type
         specified are public by default. This can lead to a
         vulnerability if a developer forgot to set the visibility
          and a malicious user is able to make unauthorized or
         unintended state changes. The {{FUNCS | comma('`')}} {{
         FUNCS | pluralize('function', 'functions')}} do not have
         a visibility set.
4  risk: 1
5  impact: 5
6  patterns:
7    - pattern: |
8        function $NAME(...) ... {
9          ...
10       }
11   - not: |
12       function $NAME(...) $VISIBILITY {
13         ...
14       }
```

I left the community to implement more rules that would allow statically finding all possible common mistakes Solidity developers tend to do.

8 Solgrep usage

The the usage of the tool is very simple. One must import the `SolGrep` class from the `solgrep` file and create a new object:

Listing 32: Importing the SolGrep utility and creating an object

```
from solgrep import SolGrep

sg = Solgrep()
```

This object can then be used to load the query files, in YAML syntax or a single solidity file.

8.1 Loading the source code

The source code can be loaded in different ways, from a file or directly from a string:

Listing 33: Source code used on this section to represent a Solidity file

```
// source.sol

pragma solidity 0.8.12;

contract Test {
    uint256 public a;

    function name() external {
        a = 1337;
    }
}
```

Loading the source code from the `source.sol` file:

Listing 34: Loading the source code from a file

```
sg.load_source_file("source.sol")
```

Loading the source code directly from a string:

Listing 35: Loading the source code from a string

```
src = '''
pragma solidity 0.8.12;

contract Test {
    uint256 public a;

    function name() external {
        a = 1337;
```

```
    }
}
'''

sg.load_source_string(src)
```

8.2 Loading the query rule

Solgrep does support multiple query formats, including a single pattern search and complex YAML pattern syntax as seen in "Solgrep rule file".

There are 4 different functions:

- `sg.load_query_file`: It will load a single file that contains a valid query solidity code.
- `sg.load_query_string`: Same as `load_query_file` but the content is taken directly from a string.
- `sg.load_query_yaml_file`: This function will load a complex yaml rule file, following the "Solgrep rule file" format.
- `sg.load_query_yaml_string`: Same as `load_query_yaml_file` but the content is taken directly from a string.

8.3 Displaying the AST

When loading a query file or source file, the parsed tree can be stored into a variable and later worked on:

Listing 36: Loading a source code and query pattern and storing the parsed trees

```
src = '''
pragma solidity 0.8.12;

contract Test {
    uint256 public a;

    function name() external {
        a = 1337;
    }
}
'''

query_src = '''
contract $CONTRACT {
    ...
}
'''

source = sg.load_source_string(src)
query = sg.load_query_string(query_src)
```

The `source` and `query` variables do contain a tree that can be printed and exported into a dot graph or png image:

Listing 37: Tree representation when printing the loaded source and query strings

```
print(source)

'source_file'
|-- 'pragma_directive'
|   |-- 'pragma'
|   |-- 'solidity_directive'
|   |   |-- 'solidity'
|   |   |-- 'pragma_versions'
|   |-- ';'
|-- 'contract_declaration'
    |-- 'contract'
    |-- 'identifier'
    |-- 'contract_body'
        |-- '{'
        |-- 'state_variable_declaration'
        |   |-- 'type_name'
        |   |   |-- 'primitive_type'
        |   |       |-- 'uint256'
        |   |-- 'visibility'
        |   |   |-- 'public'
        |   |-- 'identifier'
        |   |-- ';'
        |-- 'function_definition'
        |   |-- 'function'
        |   |-- 'identifier'
        |   |-- 'parameter_list'
        |   |   |-- '('
        |   |   |-- ')'
        |   |-- 'visibility'
        |   |   |-- 'external'
        |   |-- 'function_body'
        |       |-- '{'
        |       |-- 'assignment_expression'
        |       |   |-- 'identifier'
        |       |   |-- '='
        |       |   |-- 'number_literal'
        |       |-- ';'
        |       |-- '}'
        |-- '}'


print(query)

'source_file'
|-- 'contract_declaration'
    |-- 'contract'
    |-- 'identifier'
    |-- 'contract_body'
```

```
49          |-- '{'
50          |-- 'ellipsis'
51          |-- '}'
```

8.3.1 Exporting the AST to an image

This can be done with the following syntax witch will produce the images below the snippet:

Listing 38: Snippet showing how to export the source and query trees to an image

```
1 source.dot('root.png')
2 query.dot('query.png')
```

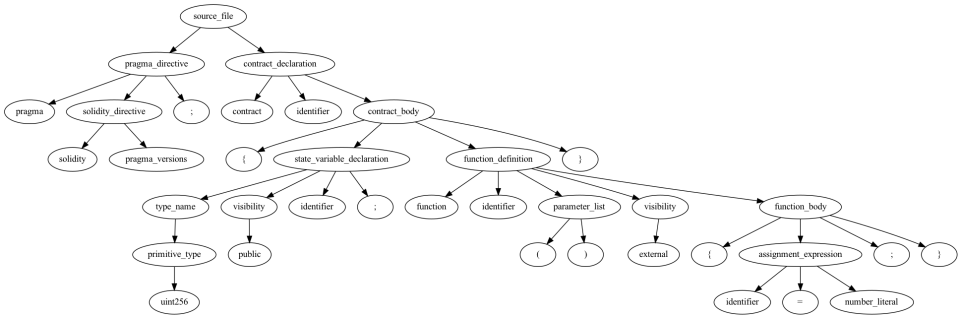

Figure 7: Tree obtained when exporting the source tree from Listing 36 (root.png)

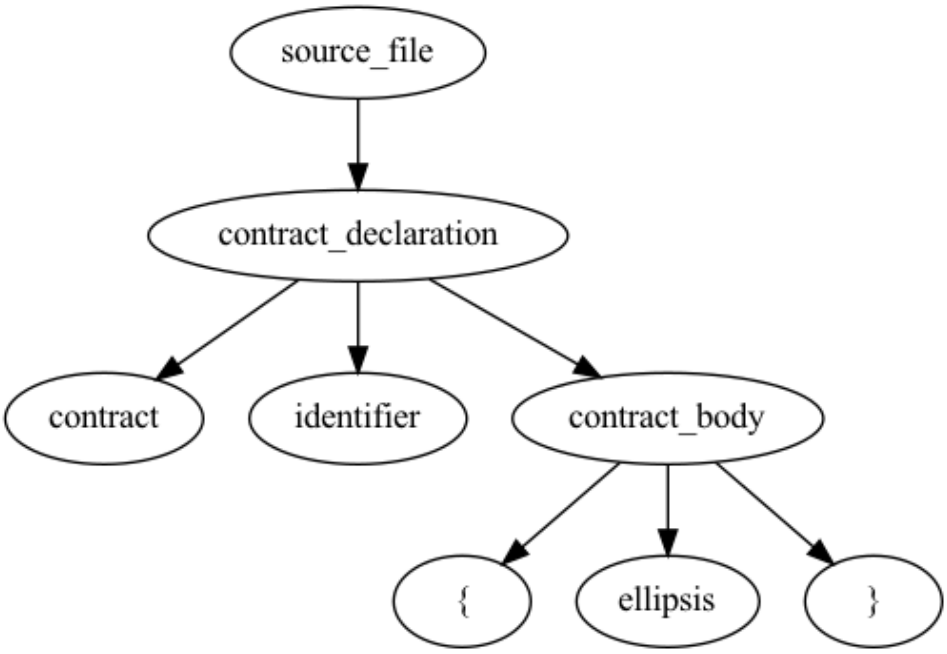

Figure 8: Tree obtained when exporting the query tree from Listing 36 (tree.png)

8.4 Getting the query results (report)

The use the `query` content against the `source` content, the `query()` function should be called. This function will compare using a BFS algorithm the nodes of both trees and skip the ones with ellipsis syntax as described on the technical section.

Listing 39: Snippet used to trigger the solgrep query

```
1 sg.query()
```

Once the query is executed, the results can be obtained by using the `report()` function, which will display a dictionary containing information about the query results.

Listing 40: Snippet used to obtain the report from the last query

```
1 report = sg.report()
```

As an example, the query and source code displayed in Listing 41 would produce the results shown in Listing 42.

Listing 41: Example file showing the usage of solgrep and how to obtain the report

```
from solgrep import SolGrep

sg = SolGrep()

src = '''
pragma solidity 0.8.12;

contract Test {
    uint256 public a;

    function name() external {
        a = 1337;
    }
}
'''

query_src = '''
id: solidity-test
message: |
  Found {{FUNCS | pluralize('a function', 'some functions')}}:
      {{FUNCS | comma}}
risk: 1
impact: 1
patterns:
  - pattern: function $FUNC(...) ... {...}
'''

sg.load_source_string(src)
sg.load_query_yaml_string(query_src)

sg.query()

report = sg.report()

print(report)
```

Listing 42: Report for the example file showcasing the solgrep usage

```
{
  "id": "solidity-test",
  "message": "Found a function: name",
  "risk": 1,
  "impact": 1,
  "results": 1,
  "metavars": [
    {
      "FUNC": [
        "name"
      ]
    }
  ],
  "bytesrange": [
    [ 68, 118 ]
```

```
16   ],
17   "linesrange": [
18     [ [ 5, 4 ], [ 7, 5 ] ]
19   ]
20 }
```

As seen in Listing 42, the returned dictionary does contain all the details for the result of the query. Including the formatted message, risk, impact, metavars lists and the `byterange` and `linesrange` of all the found results.

The `byterange` does contain the starting character and end character in the original source code that do match the query. Furthermore, the `linesrange` do contain the start line and character on that line and the end line and character of that line.

9 Conclusions

The creation of Solgrep was to expose to the security community of Solidity smart contracts a new tool which was capable of searching common mistakes made by the developers when writing smart contracts. During the project development, so many changes had to be done in order to achive the desired outcome. Although others tools do have similarities there was no real tool that would allow the flexibility that Solgrep has for Solidity. Although there is a lot to improve to the code itself and the design, I am very satisfied with the result from the point of view of the evolution that the project has undergone from the beginning to the final design.

The tasks were successfully completed in order and the dependencies between them were meet. This allowed the coding phase to move on without any obstacles allowing to achieve the final result on the coding part.

Initially, one of the objectives was to write all the SWC rules using Solgrep. However, due to time constraints, it was not possible to meet this requirement fully. Still, due to the utility that this tool would have on my daily activities, all the rules will be written and more rules out of the SWC registry will be created for me and for the community.

The initial idea of Solgrep was to be used as part of Smart Contracts Solidity Audits as a remarkable tool in the arsenal of an auditor. However, it was noticed that this tool could be easily integrated with current Solidity development stacks to find common bad patterns and coding mistakes that Solidity developers tend to do.

The tool, will always be publicly available on the https://github.com/fr0zn/solgrep repository for the Solidity Smart Contract Security to use it. Hopefully, the community would like the tool and would start contributing to the source code and create generic rules to find all possible issues related to Solidity. The tool, will be extended with newly added syntax to the Solidity programming language and will write test cases to verify the correct operation.

References

[1] "Ethereum Whitepaper." [Online]. Available: https://ethereum.org. [Accessed: 28-Sep-2021].

[2] "Solidity — Solidity 0.8.14 documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.8.14/. [Accessed: 28-May-2022].

[3] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, May 2019.

[4] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts," 18-Nov-2019.

[5] "Tree-sitter Introduction." [Online]. Available: https://tree-sitter.github.io/tree-sitter/. [Accessed: 28-Sep-2021].

[6] "Rule syntax | Semgrep." [Online]. Available: https://semgrep.dev/docs/writing-rules/rule-syntax/. [Accessed: 28-Sep-2021].

[7] "Overview · Smart Contract Weakness Classification and Test Cases." [Online]. Available: http://swcregistry.io/. [Accessed: 28-Sep-2021].

[8] "Docs home | Semgrep." [Online]. Available: https://semgrep.dev/docs/. [Accessed: 25-Feb-2022].

[9] "Pattern syntax | Semgrep." [Online]. Available: https://semgrep.dev/docs/writing-rules/pattern-syntax/. [Accessed: 28-Sep-2021].

[10] JoranHonig, *JoranHonig/tree-sitter-solidity*. 2021.

[11] "Jinja — Jinja Documentation (3.1.x)." [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x/. [Accessed: 28-Mar-2022].

[12] "ANTLR." [Online]. Available: https://www.antlr.org/. [Accessed: 28-Mar-2022].

[13] *Solidity ANTLR - Lexer*. ethereum, 2022.

[14] *Solidity ANTLR - Parser*. ethereum, 2022.

[15] "Tree-sitter Introduction | Language Bindings." [Online]. Available: https://tree-sitter.github.io/tree-sitter/#language-bindings. [Accessed: 28-Mar-2022].

[16] "Language Grammar — Solidity latest documentation." [Online]. Available: https://docs.soliditylang.org/en/latest/grammar.html. [Accessed: 28-Mar-2022].

[17] F. Celades, *Fr0zn/tree-sitter-solidity*. 2022.

[18] M. Brunsfeld, *Tree-sitter: Python bindings to the Tree-sitter parsing library*.

[19] "Any Python Tree Data — anytree 2.8.0 documentation." [Online]. Available: https://anytree.readthedocs.io/en/latest/. [Accessed: 28-Mar-2022].

[20] "Breadth-first search," *Wikipedia*. 24-Apr-2022.

[21] F. Celades, *Solgrep*. 2022.

[22] "The Official YAML Web Site." [Online]. Available: https://yaml.org/. [Accessed: 26-May-2022].

[23] "Basic Syntax | Markdown Guide." [Online]. Available: https://www.markdownguide.org/basic-syntax/. [Accessed: 27-May-2022].