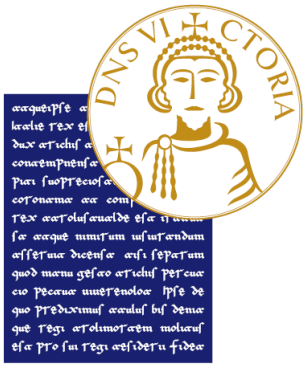

Università degli Studi del Sannio

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Linguaggi di Programmazione e Compilatori

Project Report

Solidity Metrics Extractor

Prof.ssa:
Tortorella Maria

Studenti:
Cinelli Jessica, 399000529
Mazzitelli Francesco C.,399000532
Saccone Francesco, 399000486

Anno Accademico 2022-2023

Indice

Elenco delle figure

1 Introduzione

La tecnologia Blockchain è diventata negli ultimi tempi un argomento di dibattito relativamente comune. Benchè la maggior parte dell'interesse è generato da attività legate alle criptovalute, sta diventando evidente che uno spettro molto più ampio delle applicazioni può sfruttare la tecnologia blockchain.

Un'applicazione molto diffusa della tecnologia Blockchain è quella degli *Smart Contract*: lo Smart Contract consiste nella possibilità di avere un "contratto digitale". Una delle più popolari piattaforme di blockchain che supportano gli smart contract è Ethereum.
Poiché gli smart contract in genere gestiscono denaro, è necessario garantire un numero di guasti e vulnerabilità basso. Per aiutare gli sviluppatori di smart contract per permettere lo sviluppo di queste tecnologie servono strumenti di analisi e studi sempre più sofisticati.

1.1 Obiettivo

L'obiettivo di questo project work è di analizzare gli smart contract, scritti nel linguaggio Solidity per Ethereum andando a valutare alcune metriche software. Quindi, sono state eseguite diverse attività:

- si è partiti dallo studio del linguaggio Solidity;
- una volta comprese le caratteristiche essenziali si è passati alla studio della grammatica;
- successivamente sono state definite le regole lessicali e sintattiche del linguaggio;
- infine, sono state calcolate e analizzate le metriche del software.

Le attività di generazione delle regole lessicali e sintattiche sono state eseguite con il supporto del generatore di parser ANTLR4.

2 Blockchain

La blockchain è un registro condiviso e immutabile che facilita i processi di registrazione delle transazioni e di monitoraggio degli asset nelle reti aziendali. Un asset può essere concreto (case, automobili, denaro contante, terreni) o astratto (proprietà intellettuale, brevetti, diritti d'autore, branding). Praticamente qualsiasi cosa di valore può essere monitorata e scambiata in una rete blockchain, riducendo i rischi e tagliando i costi per tutte le parti coinvolte.
La blockchain è una struttura dati che consiste in elenchi crescenti di record, denominati "blocchi", collegati tra loro in modo sicuro utilizzando la crittografia. Ogni blocco contiene un hash crittografico del blocco precedente, un timestamp e dati di transazione. Poiché ogni blocco contiene informazioni sul blocco precedente, questi formano effettivamente una catena con ogni blocco aggiuntivo che si collega a quelli precedenti. Di conseguenza, le transazioni blockchain sono irreversibili in quanto, una volta registrate, i dati in un determinato blocco non possono essere modificati retroattivamente senza alterare tutti i blocchi successivi.

2.1 Importanza della blockchain

Le attività di un business si basano sulle informazioni: più vengono ricevute in modo rapido e preciso, meglio è. La blockchain è la soluzione ideale, in quanto offre informazioni immediate, condivise e completamente trasparenti, archiviate in un registro immutabile e accessibile solamente da membri autorizzati della rete.
Una rete blockchain può monitorare ordini, pagamenti, account, processi produttivi e molto altro ancora. E dato che le informazioni affidabili sono offerte in una vista singola condivisa da tutti, sarai in grado di vedere tutti i dettagli di una transazione completa e godere di maggiore sicurezza, efficienza e nuove opportunità

La blockchain è considerata pertanto un'alternativa in termini di sicurezza, affidabilità, trasparenza e costi alle banche dati e ai registri gestiti in maniera centralizzata da autorità riconosciute e regolamentate (pubbliche amministrazioni, banche, assicurazioni, intermediari di pagamento, ecc.).

2.2 Smart contract

Gli smart contract sono semplicemente dei programmi archiviati su una blockchain che vengono eseguiti quando vengono soddisfatte delle condizioni prestabilite. Sono di norma utilizzati per automatizzare l'esecuzione di un accordo in modo che tutti i partecipanti possano essere immediatamente certi del risultato senza il coinvolgimento di intermediari o perdite di tempo. Possono anche automatizzare un flusso di lavoro, attivando l'azione successiva quando vengono soddisfatte le condizioni.

2.2.1 Funzionamento degli smart contract

Gli smart contract funzionano seguendo delle semplici istruzioni condizionali ("if/when... then... ") scritte nel codice in una blockchain. Una rete di computer esegue le azioni quando le condizioni

prestabilite sono state soddisfatte e verificate. Queste azioni potrebbero includere il rilascio di fondi alle parti appropriate, la registrazione di un veicolo, l'invio di notifiche o l'emissione di un ticket. La blockchain viene quindi aggiornata dopo che la transazione è stata completata. Ciò significa che la transazione non può essere modificata e che solo le parti a cui è stata concessa l'autorizzazione possono visualizzare i risultati.
Uno smart contract può contenere tutte le clausole necessarie per assicurare ai partecipanti che l'attività verrà completata in modo soddisfacente. Per stabilire i termini, i partecipanti devono determinare il modo in cui le transazioni e i loro dati sono rappresentati sulla blockchain, concordare le regola ipotetiche ("if/when...then...") che controllano tali transazioni, esplorare tutte le possibili eccezioni e definire una struttura per la risoluzione delle controversie.

Lo smart contract può quindi essere programmato da uno sviluppatore, sebbene le organizzazioni che utilizzano la blockchain per il business forniscano sempre di più modelli, interfacce web e altri strumenti online per semplificare la strutturazione dei contratti intelligenti.
Nella figura 1 si riporta un esempio di applicazione in campo assicurativo.

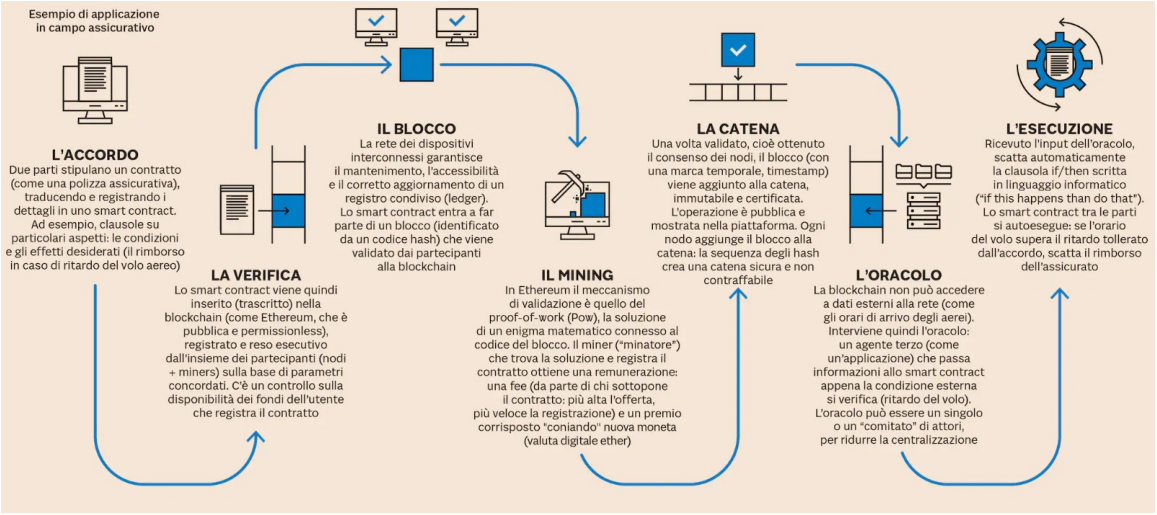

Figura 1: Esempio di applicazione in campo assicurativo

2.2.2 Vantaggi degli smart contract

- *Velocità, efficienza e accuratezza* Quando una condizione viene soddisfatta, il contratto viene eseguito immediatamente. Poiché i contratti intelligenti sono digitali e automatizzati, non ci sono documenti cartacei da elaborare né sprechi di tempo dedicato alla riconciliazione degli errori spesso risultanti dalla compilazione manuale dei documenti.

- *Attendibilità e trasparenza* Poiché non sono coinvolte terze parti e poiché i record crittografati delle transazioni sono condivisi tra i partecipanti, non è necessario chiedersi se le informazioni sono state modificate per vantaggio personale.

- *Sicurezza* I record delle transazioni della blockchain sono crittografati, il che li rende molto difficili da hackerare. Inoltre, poiché ogni record è connesso ai record precedenti e a quelli successivi in un registro distribuito, gli hacker dovrebbero modificare l'intera catena per modificare un singolo record.
- *Risparmi* I contratti intelligenti rimuovono la necessità di intermediari che gestiscano le transazioni e, per estensione, i ritardi e i costi ad essi associati.

2.3 Ethereum

Ethereum è una piattaforma basata su tecnologia blockchain open source, per cui, le operazioni realizzate vengono memorizzate in un registro pubblico. Gli utenti della rete possono creare, pubblicare, monetizzare e usare una vasta gamma di applicazioni decentralizzate con supporto per un linguaggio di programmazione Turing-complete: Solidity.

3 Analisi del linguaggio Solidity

Il primo step per la realizzazione di questo project work ha riguardato lo studio del linguaggio Solidity. Solidity è un linguaggio di alto livello orientato agli oggetti per l'implementazione di smart contract complessi, ma con modalità di definizione e codifica semplici e chiare. In paticolare, è il linguaggio principale utilizzato per la programmazione di Smart Contract su Ethereum. È influenzato da C ++, Python e JavaScript, di cui ritroviamo alcune sintassi.
Quando si distribuiscono i contratti, è necessario utilizzare l'ultima versione rilasciata versione di Solidity. A parte casi eccezionali, solo l'ultima versione riceve correzioni di sicurezza. Inoltre, i cambiamenti e Nuove funzionalità vengono introdotte regolarmente. Attualmente utilizziamo Un numero di versione 0.y.z per indicare questo rapido ritmo di cambiamento.
Poiché, come abbiamo detto prima, ogni transizione di stato viene registrata e non è modificabile, sarà necessario testare accuratamente il contratto prima di rilasciarlo in un ambiente di produzione. Le correzioni di bug possono essere costose e causare anche danni critici al sistema.

3.1 Descrizione del linguaggio

Solidity è un linguaggio di alto livello, non solo consente l'uso di operatori simbolici per indicare le operazioni e di nomi simbolici per rappresentare dati e strutture di dati, ma è anche strutturato con una sintassi e una semantica per descrivere l'algoritmo di calcolo. In particolare: Solidity è un linguaggio curly-bracket, e supporta:

- è un linguaggio curly-bracket;
- è tipizzato statisticamente;
- supporta l'ereditarietà, quindi può estendere altri smart contract;
- supporta l'utilizzo di librerie, per creare codice riutilizzabile;
- infine, supporta tipi complessi definiti dall'utente.

4 Analisi Lessicale

Per poter effettuare l'analisi lessicale sono stati definiti i token di interesse. In particolare sono stati definiti token per valori numerici, parole chiave, tipi primitivi, identificatori e operatori ed infine i token per individuare commenti e particolari sequenze di caratteri da non considerare.

4.1 Valori numerici

- **valoreNumerico** : (NumeroDecimale | NumeroEsadecimale) UnitàNumero?
- **identificatore** : ('from' | 'calldata' | Identifier)
- **VersionLiteral** : [0-9]+ '.' [0-9]+ '.' [0-9]+
- **LiteralBooleano** : 'true' | 'false'
- **NumeroDecimale** : (DigitDecimali | (DigitDecimali? '.' DigitDecimali)) ([eE] DigitDecimali)?
- ***fragment* DigitDecimali** : [0-9] ('_'? [0-9])*
- **NumeroEsadecimale** : '0' [xX] DigitEsadecimali
- ***fragment* DigitEsadecimali** : CarattereEsadecimale ('_'? CarattereEsadecimale)*
- **UnitàNumero** : 'wei' | 'szabo' | 'finney' | 'ether' | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
- **LiteralEsadecimale** : 'hex' ('"' CoppiaEsadecimale* '"' | '
  verb+'+' CoppiaEsadecimale* '
  verb+'+')
- ***fragment* CoppiaEsadecimale** : CarattereEsadecimale CarattereEsadecimale
- ***fragment* CarattereEsadecimale** : [0-9A-Fa-f]

4.2 Parole riservate

- **ReservedKeyword** : 'abstract' | 'after' | 'case' | 'catch' | 'default' | 'final' | 'in' | 'inline' | 'let' | 'match' | 'null' | 'of' | 'relocatable' | 'static' | 'switch' | 'try' | 'typeof'
- **AnonymousKeyword** : 'anonymous'
- **BreakKeyword** : 'break'
- **ConstantKeyword** : 'constant'
- **ContinueKeyword** : 'continue'
- **ExternalKeyword** : 'external'

- **IndexedKeyword** : 'indexed'
- **InternalKeyword** : 'internal'
- **PayableKeyword** : 'payable'
- **PrivateKeyword** : 'private'
- **PublicKeyword** : 'public'
- **PureKeyword** : 'pure'
- **TypeKeyword** : 'type'
- **ViewKeyword** : 'view'

4.3 Tipi primitivi

- **tipiPrimitivi** : 'address' | 'bool' | 'string' | 'var' | Int | Uint | 'byte' | Byte | Fixed | Ufixed
- **Int** : 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' | 'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' | 'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' | 'int248' | 'int256'
- **Uint** : 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' | 'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' | 'uint112' | 'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' | 'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' | 'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'
- **Byte** : 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' | 'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' | 'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' | 'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' | 'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'
- **Fixed** : 'fixed' | ('fixed' [0-9]+ 'x' [0-9]+)
- **Ufixed** : 'ufixed' | ('ufixed' [0-9]+ 'x' [0-9]+)
- **array** : tipiPrimitivi '[' ']'

4.4 Operatori e identificatori

- **Identifier** : IdentifierStart IdentifierPart*
- **incremento** : '++'
- **decremento** : '−'
- **operatoriSomma** : '+' | '-'
- **potenza** : '**'
- **operatoriMoltiplicazione** : '*' | '/' | '%'
- **operatoriRelazionali** : '<' | '>' | '<=' | '>='
- **operatoriConfrontoDiretto** : '==' | '!='
- **AND** : '&&'
- **OR** : '||'
- **NOT** : '!'
- **puntoVirgola** : ';'
- **operatoriAssegnazione** : '=' | '|=' | '^=' | '&=' | '«=' | '»=' | '+=' | '-=' | '*=' | '/=' | '%='
- ***fragment* IdentifierStart** : [a-zA-Z$_]
- ***fragment* IdentifierPart** : [a-zA-Z0-9$_]
- **LiteralStringa** : '"'CarattereDoppiApici* '"' | ''' CarattereSingoliApici*'''
- ***fragment* CarattereDoppiApici** : ∼ ["\r\n\\ | ('\\'.)
- ***fragment* CarattereSingoliApici** : ∼ ['\r\n\\] | ('\\'.)

4.5 Commenti

- **WS** : $[\backslash t \backslash r \backslash n \backslash u000C]+ \rightarrow skip$
- **COMMENTO** : '/ ∗ '.∗?' ∗ /' → *skip*
- **COMMENTO_LINEA** : '//' ∼ $[\backslash r \backslash n]* \rightarrow skip$

5 Analisi sintattica

5.1 Struttura di un file Solidity

Assunzione 1. *Un file Solidity può contenere un numero arbitrario di:*

- *pragma*
- *import*
- *definizione di contratti*

La regola di produzione associata alla struttura completa di un file Solidity è la seguente:

- sourceUnit : (pragma | imports | definizioneContratto)* EOF.

5.1.1 Pragma

La parola chiave *pragma* viene utilizzata per abilitare determinate funzionalità del compilatore. La regola di produzione associata all'utilizzo di questa direttiva è:

- pragma : 'pragma' nomePragma valorePragma puntoVirgola

  dove

  - nomePragma : identificatore
  - valorePragma : versione | expression
  - versione : vincoloVersione vincoloVersione?
  - operatoreVersione : '^' | '~' | '>=' | '>' | '<' | '<=' | '='
  - vincoloVersione : operatoreVersione? VersionLiteral

5.1.2 Import

Solidity supporta le istruzioni di import per facilitare la modularizzazione del codice.Tuttavia, non supporta il concetto di un'esportazione predefinita.
Dunque, è possibile utilizzare istruzioni import come descritto dalla regola di produzione:

- **imports** : 'import' LiteralStringa ('as' identificatore)? puntoVirgola
  | 'import' ('*' | identificatore) ('as' identificatore)? 'from' LiteralStringa puntoVirgola
  | 'import' '{' dichiarazioneImport (',' dichiarazioneImport)* '}' 'from' LiteralStringa puntoVirgola

  dove

  - dichiarazioneImport : identificatore ('as' identificatore)?

5.1.3 Definizione dei contratti

I contratti in Solidity sono simili alle classi nei linguaggi orientati agli oggetti. Ogni contratto può contenere dichiarazioni di variabili di stato, funzioni, modificatori di funzioni, eventi, errori, tipi di struttura e tipi di enumerazione. Inoltre, i contratti possono ereditare da altri contratti.

Assunzione 2. *I blocchi assembly sono gestiti come funzioni; l'accesso ai registri è considerato come identificatore.*

Assunzione 3. *fallback funcion e receive funcion sono gestite come funzioni.*

Assunzione 4. *Librerie e interfacce vengono gestite come contratti.*

La struttura generale di un contratto è descritta con la regola di produzione:

- **definizioneContratto** : ('contract' | 'interface' | 'library') identificatore ('is' inheritanceSpecifier (',' inheritanceSpecifier)*)? '{' parteDiContratto* '}'

  dove

  - **inheritanceSpecifier** : tipiDefinitiUtente ('(' listaExpression? ')')?
  - **parteDiContratto** : assegnazioneVariabile | usingForDeclaration | definizioneStruct | definizioneCostruttore | definizioneModifier | definizioneFunzione | definizioneErrore | definizioneEvento | definizioneEnum ;
  - **assegnazioneVariabile** : nomeTipo (PublicKeyword | InternalKeyword | PrivateKeyword | ConstantKeyword)* identificatore ('+=' expression)? puntoVirgola
  - **usingForDeclaration** : 'using' identificatore 'for' ('*' | nomeTipo) puntoVirgola
  - **definizioneStruct** : 'struct' identificatore '' (dichiarazioneVariabile puntoVirgola (dichiarazioneVariabile puntoVirgola)*)? '}'
  - **definizioneCostruttore** : 'constructor' listaParametri listaModifier block
  - **definizioneModifier** : 'modifier' identificatore listaParametri? block
  - **invocazioneModifier** : identificatore ('(' listaExpression? ')')?
  - **definizioneFunzione** : 'function' identificatore? listaParametri listaModifier valoreRitorno? (puntoVirgola | block)
  - **definizioneErrore** : 'error' identificatore? listaParametri puntoVirgola
  - **valoreRitorno** : 'returns' listaParametri
  - **listaModifier** : (invocazioneModifier | stateMutability | ExternalKeyword | PublicKeyword | InternalKeyword | PrivateKeyword)*
  - **definizioneEvento** :'event' identificatore listaParametriEvent AnonymousKeyword? puntoVirgola
  - **valoreEnum** : identificatore

- **definizioneEnum** : 'enum' identificatore '{' valoreEnum? (',' valoreEnum)* '}'
- **listaParametri** : '(' (parametro (',' parametro)*)? ')'
- **parametro** : nomeTipo tipoStorage? identificatore?
- **listaParametriEvent** : '(' (parametriEvent (',' parametriEvent)*)? ')'
- **parametriEvent** : nomeTipo IndexedKeyword? identificatore?
- **listaParametriFunzione** : '(' (parametroFunzione (',' parametroFunzione)*)? ')'
- **parametroFunzione** : nomeTipo tipoStorage?
- **dichiarazioneVariabile** : nomeTipo tipoStorage? identificatore
- **nomeTipo** : tipiPrimitivi | tipiDefinitiUtente | mapping | nomeTipo '[' expression? ']' | functionTypeName | 'address' 'payable'
- **tipiDefinitiUtente** : identificatore ('.' identificatore)*
- **mapping** : 'mapping' '(' tipiPrimitivi '=>' nomeTipo ')'
- **functionTypeName** : 'function' listaParametriFunzione (InternalKeyword | ExternalKeyword | stateMutability)* ('returns' listaParametriFunzione)?

  tipoStorage : 'memory' | 'storage' | 'calldata'
- **stateMutability** : PureKeyword | ConstantKeyword | ViewKeyword | PayableKeyword

5.2 Strutture di controllo ed espressioni

La maggior parte delle strutture di controllo conosciute dai linguaggicurly-braces sono disponibili in Solidity: ifelsewhiledoforbreakcontinuereturn Le parentesi sono obbligatorie per racchiudere le condizioni, ma non sono necessarie per il corpo dello statement.

- **block** : '{' statement* '}' ;
- **statement** : ifStatement | whileStatement | forStatement | block | doWhileStatement | continueStatement | breakStatement | returnStatement | throwStatement | emitStatement | simpleStatement
- **expressionStatement** : expression puntoVirgola
- **ifStatement** : 'if' '(' expression ')' statement ('else' statement)?
- **whileStatement** : 'while' '(' expression ')' statement
- **simpleStatement** : (dichiarazioneVariabileStatement | expressionStatement)
- **forStatement** : 'for' '(' (simpleStatement | puntoVirgola) (expressionStatement | puntoVirgola) expression? ')' statement
- **doWhileStatement** : 'do' statement 'while' '(' expression ')' puntoVirgola

- **continueStatement** : 'continue' puntoVirgola
- **breakStatement** : 'break' puntoVirgola
- **returnStatement** : 'return' expression? puntoVirgola
- **throwStatement** : 'throw' puntoVirgola
- **emitStatement** : 'emit' chiamataFunzione puntoVirgola
- **tryCatchStatement** : 'try' expression statement listaCatch
- **listaCatch** : 'catch' (expression | '(' expression ')') statement | 'catch' (expression | '(' expression ')') statement listaCatch
- **dichiarazioneVariabileStatement** : ('var' listaIdentifier | dichiarazioneVariabile | '(' listaDichiarazioneVariabili ')') ('=' expression)? puntoVirgola
- **listaDichiarazioneVariabili** : dichiarazioneVariabile? (',' dichiarazioneVariabile?)*
- **listaIdentifier** : '(' (identificatore? ',')* identificatore? ')'

Per quanto riguarda le espressioni, l'ordine di valutazione delle espressioni non è specificato; più formalmente, non è specificato l'ordine in cui vengono valutati i figli di un nodo nell'albero delle espressioni, ma sono ovviamente valutati prima del nodo stesso.

Assunzione 5. *La struttra dati array è gestita come un'espressione.*

- **expression** : expression (incremento | decremento)
  | 'new' nomeTipo
  | expression '[' expression ']'
  | expression '(' argomentiChiamataFunzione ')'
  | expression '.' identificatore
  | '(' expression ')'
  | (incremento | decremento) expression
  | (operatoriSomma) expression
  | ('after' | 'delete') expression
  | NOT+ expression
  | '∼' expression
  | expression potenza expression
  | expression (operatoriMoltiplicazione) expression
  | expression (operatoriSomma) expression
  | expression ('«' | '»') expression
  | expression '&' expression
  | expression '^' expression
  | expression '|' expression
  | expression (operatoriRelazionali) expression

| expression (operatoriConfrontoDiretto) expression
| expression AND expression
| expression OR expression
| expression '?' expression ':' expression
| expression (operatoriAssegnazione) expression
| primaryExpression

- **primaryExpression** : LiteralBooleano | valoreNumerico | LiteralEsadecimale | LiteralStringa | identificatore ('[' ']')? | TypeKeyword | tupleExpression | tipiExpression ('[' ']')?
- **listaExpression** : expression (',' expression)*
- **listaNameValue** : nameValue (',' nameValue)* puntoVirgola?
- **nameValue** : identificatore ':' expression
- **argomentiChiamataFunzione** : '' listaNameValue? '' | listaExpression?
- **chiamataFunzione** : expression '(' argomentiChiamataFunzione ')'
- **tupleExpression** : '(' (expression? (',' expression?)*) ')' | '[' (expression (',' expression)*)? ']'
- **tipiExpression** : tipiPrimitivi | tipiDefinitiUtente

6 Analisi semantica

Il tool ANTRL non consente di svolgere un'analisi semantica articolata direttamente nel file di specifica. Per questo motivo, l'analisi semantica è stata spostata fuori dal file .g4 tramite la creazione di package appositi, ognuno dei quali identifica una parte specifica dell'analisi semantica. In particolare, l'analisi semantica è stata suddivisa nelle seguenti fasi:

1. *Analisi lessicale e sintattica degli smart contract*
2. *Creazione e visualizzazione dell'albero di derivazione*
3. *Creazione della tabella dei simboli*
4. *Calcolo delle metriche*

Nelle prossime sezioni verranno analizzate le singole fasi nel dettaglio.

6.1 Analisi lessicale e sintattica degli smart contract

Questa è la fase relativa all'analisi dei file sorgente degli smart contract.
Per ogni file sorgente viene creato uno stream di caratteri da analizzare; tale stream di caratteri viene convertito in uno stream di token grazie all'utilizzo del *Lexer*. Successivamente si passa da uno stream di token ad un albero di derivazione astratto con l'ausilio del *Parser*.

```
List<String> solPaths = new ArrayList<>();
File[] contents = input_dir.listFiles();

if (contents.length > 0) {
    for(File content : contents) {
        if(content.isFile()) solPaths.add(content.toString());
    }
}

if (solPaths.isEmpty()) {
    throw new ParseException("Directory contenente i file vuota!", 0);
}

try (BufferedWriter writer =
    new BufferedWriter(new FileWriter(output_dir+"\\metrics.csv"));
    CSVPrinter csvPrinter = new CSVPrinter(writer,
        CSVFormat.EXCEL.withHeader("SolidityFile", "ContractName", "Type",
                "SLOC", "CLOC", "McCC", "NOS", "CBO", "DIT"))) {

    for (String solPath : solPaths) {
        String contractCode = read(solPath, Charset.forName("UTF-8"));
        CharStream charStream = CharStreams.fromString(contractCode);
        SymbolTable table = new SymbolTable();
        TokenSource lexer = new SolidityLexer(charStream);
        BufferedTokenStream tokens = new BufferedTokenStream(lexer);
        SolidityParser parser = new SolidityParser(tokens);
```

6.2 Creazione e visualizzazione dell'albero di derivazione

La seconda fase consente di costruire l'albero di derivazione e di visualizzarne il contenuto al fine di rendere maggiormente comprensibili le regole di derivazione e la loro interazione.

```
MAIN
TreePlotter.getRules(parser);
TreePlotter.createParseTree(parser);
TreePlotter.plot();
parser.reset();
```

```
TREE PLOTTER
public class TreePlotter {

   /** Rilevatore End-Of-Line */
   public static final String Eol = System.lineSeparator();

   /** Carattere d'indentazione utilizzato per il pretty-printing */
   public static final String Indents = " ";
   private static int level;

   private static List<String> ruleNamesList;
   private static ParseTree tree;

   public TreePlotter(){}

   public static void createParseTree(SolidityParser parser) {
    tree = parser.sourceUnit();
}

   public static void getRules(SolidityParser parser) {
    ruleNamesList = Arrays.asList(parser.getRuleNames());
   }

   //Pretty print di un intero albero. getNodeText viene utilizzato per
   //ottenere il contenuto del nodo. (Trees.toStringTree(....))
   public static String toPrettyTree(final Tree t,
              final List<String> ruleNames) {
       level = 0;
       return process(t, ruleNames)
              .replaceAll("(?m)^\\s+$", "")
              .replaceAll("\\r?\\n\\r?\\n", Eol);
   }

   private static String process(final Tree t, final List<String> ruleNames) {
       if (t.getChildCount() == 0) return Utils
              .escapeWhitespace(Trees.getNodeText(t, ruleNames), false);
       StringBuilder sb = new StringBuilder();
       sb.append(lead(level));
       level++; //incrementa numero di indentazioni
       String s = Utils.escapeWhitespace(Trees.getNodeText(t, ruleNames), false);
       sb.append(s + ' ');
```

```
        for (int i = 0; i < t.getChildCount(); i++) {
            //chiamata ricorsiva al metodo
            sb.append(process(t.getChild(i), ruleNames));
        }
        level--; //decrementa numero di indentazioni
        sb.append(lead(level));
        return sb.toString();
    }

    //Metodo adibito all'indentazione
    private static String lead(int level) {
        StringBuilder sb = new StringBuilder();
        if (level > 0) {
            sb.append(Eol);
            for (int cnt = 0; cnt < level; cnt++) {
                sb.append(Indents);
            }
        }
        return sb.toString();
    }

    //Metodo adibito alla stampa dell'albero di derivazione
    public static void plot() {
     System.out.println(toPrettyTree(tree, ruleNamesList));
    }
}
```

Esempio di pretty printing dell'albero di derivazione dello script Ballot.sol:

```
    pragma pragma
        nomePragma
          identificatore solidity
        valorePragma
          versione
            vincoloVersione
              operatoreVersione >=
              0.7.0
            vincoloVersione
              operatoreVersione <
              0.9.0
        puntoVirgola ;
  definizioneContratto contract
    identificatore Ballot
    {
    parteDiContratto
      definizioneStruct struct
        identificatore Voter
        {
        dichiarazioneVariabile
          nomeTipo
            ... -> EOF
```

6.3 Creazione della tabella dei simboli

Altra fase fondamentale dell'analisi sintattica è stata la creazione della tabella dei simboli, struttura dati utilizzata con lo scopo di mantenere tutte le informazioni relative alla dichiarazione di identificatori. La struttura dati che meglio si presta come base per la creazione di una tabella dei simboli è una Mappa. In Java tale struttura è un interfaccia che viene implementata dalle classi TreeMap e HashMap. Siccome non è stato ritenuto necessario che la Mappa fosse ordinata, è stata preferita una HashMap.
Gli elementi vengono salvati nella Mappa costruendo una "tupla" di tipo chiave-valore in cui la chiave rappresenta il nome della variabile o della funzione definita e il valore la tipologia di Token, in questo caso "identificatore" La tabella mantiene informazioni relative ad identificatori generati durante la fase di dichiarazione delle variabili ed identificatori generati durante la fase di dichiarazione delle funzioni

```
MAIN
List<String> ruleNamesList = Arrays.asList(parser.getRuleNames());
ParseTree tree = parser.sourceUnit();
table.populateMap(tree, ruleNamesList);
table.printTable();
parser.reset();
```

```
SYMBOL TABLE
public class SymbolTable {

private static HashMap<String, String> table;

 public SymbolTable() {
  SymbolTable.table = new HashMap<String, String>();
 }

 public static void addSymbol(String id, String symbol) {
  if(!table.containsKey(id))
   table.put(id, symbol);
 }

 public static void modifyValue(String id, String symbol) {
  table.remove(id);
  if(!table.containsKey(id))
   table.put(id, symbol);
 }

    public boolean populateMap(final Tree t, final List<String> ruleNames) {

        boolean flag = false;

        if (t.getChildCount() == 0) return false;

        Stack<Tree> stack = new Stack<>();
        stack.push(t);
        while(stack.empty() == false) {
```

```
          Tree t0 = stack.pop();
          for(int i = 0; i < t0.getChildCount(); i++) {
              stack.push(t0.getChild(i));
              String value = Trees.getNodeText(t0, ruleNames);
              if(value.equals("identificatore") && flag == false) {
                   addSymbol(Trees.getNodeText(stack.pop(), ruleNames), value);
                   flag = true;
               }
          }
          flag = false;
       }
    return true;
    }

 public static String getSymbol(String id) {
  return table.get(id);
 }
 public static boolean contains(String id) {
  if(table.containsKey(id)) return true;
  else return false;
 }

 public void printTable() {
  Set<String> keys =table.keySet();
  for(String key:keys) {
   System.out.println("ID:"+key+"|VALUE:"+table.get(key));

  }
 }
}
```

Esempio di tabella dei simboli dello script Ballot.sol:

```
   ID: msg              |      VALUE: identificatore
   ID: winnerName       |      VALUE: identificatore
   ID: voters           |      VALUE: identificatore
   ID: voted            |      VALUE: identificatore
   ID: winningProposal  |      VALUE: identificatore
   ID: delegate         |      VALUE: identificatore
   ID: winningVoteCount |      VALUE: identificatore
   ID: voteCount        |      VALUE: identificatore
   ID: to               |      VALUE: identificatore
   ID: voter            |      VALUE: identificatore
   ID: proposalNames    |      VALUE: identificatore
   ID: winningProposal_ |      VALUE: identificatore
   ID: giveRightToVote  |      VALUE: identificatore
   ID: Proposal         |      VALUE: identificatore
```

6.4 Calcolo delle metriche

L'ultima fase riguarda il calcolo delle metriche. Tramite l'iterazione dell'albero di derivazione, vengono calcolate delle metriche associate a caratteristiche qualitative del sorgente. Nel momento in cui viene riconosciuta una regola di produzione verrà eseguita una particolare azione semantica. Per poter implementare un'azione semantica relativa ad una regola di produzione è necessario:

- Che venga dapprima implementata l'interfaccia "Visitor" che viene autogenerata dal file di specifica
- La classe che implementa tale interfaccia, chiamata "BaseVisitor" viene anch'essa autogenerata dal file di specifica costruendo un vero e proprio framework da specializzare
- E infine l'estensione della classe "framework" in modo da poterne specializzare i metodi

Ogni classe deve definire le azioni semantiche da eseguire nel momento in cui viene riconosciuta la regola di produzione, segue un esempio relativo ai metodi fondamentali di BaseVisitor.

```
// Generated from Solidity.g4 by ANTLR 4.7.1

public class SolidityBaseVisitor<T> extends AbstractParseTreeVisitor<T>
       implements SolidityVisitor<T> {

 @Override public T visitSourceUnit(SolidityParser.SourceUnitContext ctx) {
           return visitChildren(ctx);
           }

 @Override public T visitPragma(SolidityParser.PragmaContext ctx) {
           return visitChildren(ctx);
           }

 @Override public T visitImports(SolidityParser.ImportsContext ctx) {
           return visitChildren(ctx);
           }

 @Override public T visitDefinizioneContratto(
           SolidityParser.DefinizioneContrattoContext ctx) {
           return visitChildren(ctx);
           }
}
```

6.4.1 Complessità ciclomatica di McCabe

Si riporta di seguito la classe relativa alla logica del calcolo della metrica relativa alla complessità ciclomatica di McCabe. Come si può notare dal codice, la variabile mcc viene incrementata ogni volta che, durante l'iterazione, si incontra uno statement relativo a cicli e condizioni

```
public class McCabeMetricExtractor extends SolidityBaseVisitor<Integer> {

    private int mcc;

    public McCabeMetricExtractor() {
    mcc = 0;
    }

    @Override
    public Integer visitDefinizioneFunzione(DefinizioneFunzioneContext ctx) {
        super.visitDefinizioneFunzione(ctx);
        return mcc + 1;
    }

    @Override
    public Integer visitIfStatement(IfStatementContext ctx) {
        mcc++;
        super.visitIfStatement(ctx);
        return null;
    }

    @Override
    public Integer visitWhileStatement(WhileStatementContext ctx) {
        mcc++;
        super.visitWhileStatement(ctx);
        return null;
    }

    @Override
    public Integer visitForStatement(ForStatementContext ctx) {
        mcc++;
        super.visitForStatement(ctx);
        return null;
    }

    @Override
    public Integer visitDoWhileStatement(DoWhileStatementContext ctx) {
        mcc++;
        super.visitDoWhileStatement(ctx);
        return null;
    }
}
```

6.4.2 Metriche Object Oriented

Per ogni smart contract sono state calcolate, oltre la complessità ciclomatica di McCabe, le seguenti metriche object oriented:

- **Source Lines Of Code (SLOC)**:misura le dimensioni di un software basandosi sul numero di linee di codice sorgente;
- **Comments Line Of Code (CLOC)**: misura le dimensioni dei commenti di documentazione associati al sorgente;
- **Number of Statements (NOS)**: conta il numero di statement presenti nel file;
- **Coupling Between Object Classes (CBO)**: riassume gli accoppiamenti che accomuna una classe con le altre classi appartenenti alla struttura interna dell'applicazione;
- **Depth of Inheritance Tree (DIT)**: Indica la distanza massima di un nodo (o classe) dalla radice dell'albero rappresentante la struttura ereditaria;
- **Numero di contratti per file (N_contr)**: conta il numero di contratti presenti in un file .SOL;
- **Numero di librerie per file (N_lib)**: conta il numero di librerie presenti in un file .SOL;
- **Numero di interfacce per file (N_inter)**: conta il numero di interfacce presenti in un file .SOL;
- **Numero totale di elementi (N_totElem)**: conta il numero di elementi (librerie, interfacce e contratti) presenti in un file .SOL.

Queste metriche sono state calcolate utilizzando lo stesso meccanismo adottato per il calcolo della complessità ciclomatica di McCabe descritto nella sezione 6.4.1.

7 Risultati

Nell'analizzare gli smart contract e calcolarne quindi le metriche descritte precedentemente, si è giunti a delle conclusioni.
In particolare, sono stati analizzati 144 smart contract, ognuno dei quali si trova nei seguenti script Solidity:

- **AhooleeTokenPreSale**
- **Ballot**
- **BasicToken**
- **BatchMintMetadata**
- **BlindAuction**
- **BLTTokenSale**
- **Mock**
- **Purchase**
- **ReceiverPays**
- **StoxSmartTokenSale**
- **TWStrings**

Dall'analisi delle seguenti classi, è stato prodotto un file CSV contenente i risultati del calcolo delle metriche OO ricavate nella fase precedente.

Viene qui di seguito riportata una breve anteprima del file con gli outputs.

SolidityFile	ContractName	Type	SLOC	CLOC	McCC	NOS	CBO	DIT	N_contr	N_lib	N_inter	N_totElem
AhooleeTokenPreSale.sol	SafeMath	library	42	2	8	0	0	0	8	1	0	9
AhooleeTokenPreSale.sol	AhooleeToken	contract	16	3	1	0	0	3	8	1	0	9
AhooleeTokenPreSale.sol	Haltable	contract	24	2	2	6	0	1	8	1	0	9
AhooleeTokenPreSale.sol	ERC20	contract	6	0	3	0	0	1	8	1	0	9
AhooleeTokenPreSale.sol	ERC20Basic	contract	6	0	2	0	0	0	8	1	0	9
AhooleeTokenPreSale.sol	StandardToken	contract	51	23	4	0	0	2	8	1	0	9
AhooleeTokenPreSale.sol	BasicToken	contract	36	13	2	3	0	1	8	1	0	9
AhooleeTokenPreSale.sol	AhooleeTokenPreSale	contract	126	0	18	6	1	2	8	1	0	9
AhooleeTokenPreSale.sol	Ownable	contract	21	0	3	3	0	0	8	1	0	9
Ballot.sol	Ballot	contract	151	60	9	4	0	0	1	0	0	1
BasicToken.sol	SafeMath	library	25	2	4	0	0	0	5	1	0	6
BasicToken.sol	ERC20Basic	contract	6	0	2	0	0	0	5	1	0	6
BasicToken.sol	StandardToken	contract	72	27	6	0	0	2	5	1	0	6
BasicToken.sol	BasicToken	contract	31	11	2	0	0	1	5	1	0	6
BasicToken.sol	SimpleToken	contract	17	3	1	0	0	3	5	1	0	6
BasicToken.sol	ERC20	contract	6	0	3	0	0	1	5	1	0	6

Figura 2: Esempio Output CSV

Dall'analisi dunque è possibile stabilire:

- **Due contratti hanno un valore di CBO superiore al valore di soglia (>2)**. Valori elevati del CBO complicano testing e modifiche.
- **Diversi contratti hanno un numero di linee di codice eccessivo**
- **Diversi contratti hanno un numero di linee di commenti ridotto.** Questa non è considerata una buona pratica nello sviluppare codice.
- **Come risultato positivo, è emerso che nessun contratto ha un valore eccessivo alla soglia di DIT (>7).** In particolare più è alto il valore, maggiore è il numero di metodi che eredita, rendendo più complesso prevederne il comportamento.
- **Il file .SOL tra quelli analizzati con il numero di contratti per file è TWStrings.** Nel dettaglio è possibile analizzare inoltre che questo file è quello che presenta più librerie tra tutti gli altri.
- **Un risultato interessante emerso nell'analisi degli smart contract è che in nessun file .SOL sono presenti interfacce.**

8 Commenti e Link

- Per la stesura del seguente report è stato usato il linguaggio di marcatura per la preparazione di testi LaTeX.
- Come sistema di hosting per il repository del progetto è stato usato GitHub.

Link GitHub del progetto: https://github.com/FrancescoMazzitelli/SolidityMetricsExtractor