ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παραγωγή Σημασιολογικά Ορθών Προγραμμάτων Solidity για τον Αυτόματο Έλεγχο του Μεταγλωττιστή της

Συγγραφέας:
Αλέξανδρος Κοντογιάννης

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Σεπτέμβριος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παραγωγή Σημασιολογικά Ορθών Προγραμμάτων Solidity για τον Αυτόματο Έλεγχο του Μεταγλωττιστή της

Συγγραφέας:
Αλέξανδρος Κοντογιάννης

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Σεπτεμβρίου 2025

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
Κωνσταντίνος Σαγώνας Αναπληρωτής Καθηγητής ΕΜΠ	Νικόλαος Παπασπύρου Καθηγητής ΕΜΠ	Ζωή Παρασκευοπούλου Επίκουρη Καθηγήτρια ΕΜΠ

Αθήνα, Σεπτέμβριος 2025

Αλέξανδρος Κοντογιάννης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Η διασφάλιση της ορθότητας ενός μεταγλωττιστή είναι ένα απαιτητικό αλλά απαραίτητο έργο, ιδίως καθώς οι σύγχρονοι μεταγλωττιστές αυξάνονται σε μέγεθος και πολυπλοκότητα. Μεταξύ των διαφόρων τεχνικών, η παραγωγή τυχαίων προγραμμάτων έχει αποδειχθεί αποτελεσματική στην ανίχνευση σφαλμάτων μεταγλωττιστή που τα παραδοσιακά σετ ελέγχων ενδέχεται να παραβλέπουν. Παράλληλα, με την αυξανόμενη δημοτικότητα πλατφορμών blockchain όπως το Ethereum, η γλώσσα προγραμματισμού Solidity έχει καταστεί ευρέως χρησιμοποιούμενο εργαλείο για την ανάπτυξη έξυπνων συμβολαίων. Ο μεταγλωττιστής της Solidity, ο solc, είναι ακόμη σχετικά νέος και συνεχίζει να εξελίσσεται, γεγονός που τον καθιστά επιρρεπή σε σφάλματα και παλινδρομήσεις (regressions). Η παρούσα εργασία παρουσιάζει το SolGen, μία γεννήτρια τυχαίων σημασιολογικά ορθών προγραμμάτων Solidity, με στόχο τον έλεγχο του μεταγλωττιστή της Solidity.

Λέξεις κλειδιά: μεταγλωττιστές, ορθότητα μεταγλωττιστών, έλεγχος μεταγλωττιστών, παραγωγή τυχαίων προγραμμάτων, Solidity

Abstract

Ensuring compiler correctness is a challenging but essential task, particularly as modern compilers grow in size and complexity. Among various techniques, random program generation has proven effective at uncovering subtle compiler bugs that traditional test suites may overlook. In parallel, with the growing popularity of blockchain platforms such as Ethereum, the Solidity programming language has become a widely used tool for developing smart contracts. The Solidity compiler, solc, is still relatively new and continues to evolve, making it susceptible to bugs and regressions. This work presents SolGen, a generator of semantically correct Solidity programs which aims to test the Solidity compiler.

Keywords: compilers, compiler correctness, compiler testing, random program generation, Solidity

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Κωστή Σαγώνα, για την υποστήριξη, την καθοδήγηση και την ευκαιρία που μου έδωσε να ασχοληθώ με το αντικείμενο του ενδιαφέροντός μου, αυτό των Μεταγλωττιστών.

Επίσης, είμαι ευγνώμων προς την οικογένειά μου για τη συνεχή και άνευ όρων στήριξή της, χωρίς την οποία δεν θα ήταν δυνατή η ολοκλήρωση των σπουδών μου και η εκπόνηση της παρούσας εργασίας.

Αλέξανδρος Κοντογιάννης
Αθήνα, Σεπτέμβριος 2025

Contents

Κεφάλαιο 1

Εκτενής Περίληψη στα Ελληνικά

1.1 Εισαγωγή

1.1.1 Κίνητρο

Οι μεταγλωττιστές συχνά θεωρούνται απαλλαγμένοι από σφάλματα και, γενικά, τυγχάνουν εμπιστοσύνης ότι λειτουργούν όπως θα έπρεπε. Ωστόσο, ένας μεταγλωττιστής είναι ο ίδιος ένα πρόγραμμα και, ως εκ τούτου, μπορεί να περιέχει σφάλματα. Δεδομένου του κρίσιμου ρόλου που διαδραματίζουν οι μεταγλωττιστές στην ανάπτυξη λογισμικού, η διασφάλιση της ορθότητάς τους αποτελεί ενεργό και διαρκές πεδίο έρευνας. Οι συνήθεις προσεγγίσεις για την αντιμετώπιση του προβλήματος βασίζονται κυρίως σε εκτεταμένο έλεγχο, συμπεριλαμβανομένων σύγχρονων τεχνικών fuzzing και τυχαίας παραγωγής προγραμμάτων [1]. Παράλληλα, έχουν διερευνηθεί τεχνικές τυπικής επαλήθευσης—όπως εκείνες που χρησιμοποιούνται στο CompCert—με στόχο την μαθηματική απόδειξη της ορθότητας του μεταγλωττιστή [2].

Οι μεταγλωττιστές υποβάλλονται σε εντατικούς ελέγχους για την επαλήθευση ότι συμπεριφέρονται όπως αναμένεται. Οι προγραμματιστές των μεταγλωττιστών δημιουργούν χειροκίνητα μεγάλα σύνολα ελέγχου που καλύπτουν ένα ευρύ φάσμα γλωσσικών δομών. Παρότι αυτό παρέχει μια βασική εμπιστοσύνη στη λειτουργικότητα του μεταγλωττιστή, η κατασκευή τέτοιων συνόλων ελέγχου είναι επίπονη διαδικασία και οι εγγενείς μεροληψίες των δημιουργών τους μπορεί να οδηγήσουν σε κενά, καθώς δεν είναι πάντα δυνατό να προβλεφθούν όλες οι ασυνήθιστες ή οριακές χρήσεις των γλωσσικών δομών. Εκεί είναι που συχνά παραμένουν κρυμμένα τα σφάλματα μεταγλωττιστών. Αυτό έχει οδηγήσει σε προσπάθειες τυχαίας παραγωγής προγραμμάτων, οι οποίες μετριάζουν ορισμένους περιορισμούς των χειροποίητων συνόλων ελέγχου.

Η Solidity είναι μία γλώσσα προγραμματισμού σχεδιασμένη ειδικά για την ανάπτυξη έξυπνων συμβολαίων που εκτελούνται στο Ethereum Virtual Machine (EVM) [3]. Διαφέρει από άλλες

γλώσσες, καθώς οφείλει να παρέχει δυνατότητες και λειτουργίες που σχετίζονται με έννοιες ειδικές για έξυπνα συμβόλαια και για το EVM. Επειδή τα έξυπνα συμβόλαια της Solidity διαχειρίζονται πραγματικά περιουσιακά στοιχεία και εκτελούνται μέσα σε μη αναστρέψιμες συναλλαγές, ακόμη και ανεπαίσθητες λανθασμένες μεταγλωττίσεις μπορεί να έχουν συνέπειες ασφάλειας—προκαλώντας εσφαλμένες ενημερώσεις κατάστασης, λανθασμένες μεταφορές ή παρακάμψεις ελέγχων. Ο μεταγλωττιστής της Solidity, solc, είναι συγκριτικά νέος και εξελίσσεται μαζί με τη γλώσσα και το EVM, και προηγούμενες εκδόσεις του έχουν αποκαλύψει ποικίλα σφάλματα. Ο συνδυασμός του εξειδικευμένου μοντέλου εκτέλεσης και του περιβάλλοντος ανάπτυξης υψηλού ρίσκου καθιστά τον αυστηρό έλεγχο ιδιαίτερα σημαντικό για την καθιέρωση της ορθότητας του μεταγλωττιστή.

1.1.2 Στόχος

Στόχος της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη μίας γεννήτριας τυχαίων, σημασιολογικά ορθών προγραμμάτων Solidity, με σκοπό τον έλεγχο του μεταγλωττιστή της και την πιθανή αποκάλυψη σφαλμάτων. Έχουν υπάρξει και άλλες προσπάθειες fuzzing του solc, κυρίως από AFL-based fuzzers [4, 5, 6, 7], όμως, σύμφωνα με ό,τι γνωρίζουμε, δεν έχει αναπτυχθεί αντίστοιχο generation-based εργαλείο για τη Solidity. Η ομάδα της Solidity έχει αναπτύξει έναν σημασιολογικό fuzzer [8], αλλά ο στόχος του είναι ο έλεγχος του σταδίου βελτιστοποίησης του ενδιάμεσου κώδικα του μεταγλωττιστή και λειτουργεί στο επίπεδο της ενδιάμεσης αναπαράστασης της γλώσσας, γνωστής ως Yul.

1.1.3 Περιορισμοί

Η τυχαία παραγωγή προγραμμάτων για μία ολόκληρη γλώσσα προγραμματισμού είναι μία σύνθετη και χρονοβόρα διαδικασία που υπερβαίνει τους σκοπούς της παρούσας εργασίας. Ως εκ τούτου, η γεννήτρια περιορίζεται σε ένα υποσύνολο της Solidity, ενώ οι μη υλοποιημένες δομές αποτελούν κατεύθυνση για μελλοντική εργασία.

Η γεννήτρια προγραμμάτων έχει υλοποιηθεί ειδικά για τη Solidity και οι σημασιολογικοί περιορισμοί είναι κωδικοποιημένοι στο εσωτερικό της. Αν και θα μπορούσε να δημιουργηθεί ένα πιο γενικό εργαλείο που να δέχεται τους σημασιολογικούς περιορισμούς ως είσοδο, αυτό δεν αποτέλεσε στόχο της παρούσας εργασίας.

Τέλος, η ανίχνευση μη επιθυμητής συμπεριφοράς στον μεταγλωττιστή περιορίζεται αποκλειστικά στον χρόνο μεταγλώττισης, όπως τονίζεται στην Ενότητα 1.3.

1.1.4 Συνεισφορές

Με τη χρήση του SolGen εντοπίσαμε σφάλματα σε διαφορετικές εκδόσεις του μεταγλωττιστή της Solidity, με ορισμένα από αυτά να βρίσκονται στις πιο πρόσφατες εκδόσεις· κάποια μάλιστα ήταν προηγουμένως άγνωστα και αναφέρθηκαν στους προγραμματιστές της Solidity. Η κύρια συνεισφορά της εργασίας δεν έγκειται στην ποσότητα των σφαλμάτων που βρέθηκαν, αλλά

στην παροχή ενός εργαλείου που μπορεί να μειώσει την ανάγκη για χειροκίνητη κατασκευή προγραμμάτων ελέγχου.

1.1.5 Δομή του Κεφαλαίου

Οι υπόλοιπες ενότητες του κεφαλαίου οργανώνονται ως εξής:

Στην επόμενη Ενότητα, δίνουμε μια συνοπτική επισκόπηση βασικών εννοιών και χαρακτηριστικών της Solidity σχετικών με την παρούσα εργασία.

Στην Ενότητα 1.3, εισάγουμε το SolGen, την προτεινόμενη γεννήτρια τυχαίων προγραμμάτων και εξετάζουμε τη διαδικασία παραγωγής για διάφορες γλωσσικές δομές.

Στην Ενότητα 1.4, αξιολογούμε το SolGen χρησιμοποιώντας μετρικές κάλυψης κώδικα.

Στην Ενότητα 1.5, συνοψίζουμε αποτελέσματα, περιορισμούς και κατευθύνσεις για μελλοντική εργασία.

1.2 Solidity

Η Solidity είναι μια γλώσσα προγραμματισμού με στατικό σύστημα τύπων, σχεδιασμένη ειδικά για την ανάπτυξη έξυπνων συμβολαίων που εκτελούνται στο Ethereum Virtual Machine (EVM). Το EVM είναι το αποκεντρωμένο περιβάλλον εκτέλεσης που είναι υπεύθυνο για την εκτέλεση έξυπνων συμβολαίων στο Ethereum blockchain. Σε αντίθεση με γλώσσες γενικού σκοπού, η Solidity είναι προσαρμοσμένη στις ιδιαίτερες απαιτήσεις των εφαρμογών blockchain, προσφέροντας δυνατότητες όπως η αποστολή και λήψη Ether (το εγγενές κρυπτονόμισμα του Ethereum), ο χειρισμός σφαλμάτων ειδικά για συναλλαγές (π.χ. αναίρεση συναλλαγής), καθώς και ρητό έλεγχο πάνω σε διακριτές περιοχές δεδομένων μέσα στο EVM (`storage`, `memory`, `calldata`). Παρότι μια πληρέστερη περιγραφή της γλώσσας βρίσκεται στην επίσημη τεκμηρίωσή της, η παρούσα ενότητα παρέχει μια επισκόπηση βασικών δομών και συμπεριφορών της γλώσσας που είναι σχετικές με αυτή τη διπλωματική εργασία.

1.2.1 Δομή ενός Έξυπνου Συμβολαίου

Τα έξυπνα συμβόλαια αποτελούν τα δομικά στοιχεία ενός αρχείου πηγαίου κώδικα Solidity. Τα έξυπνα συμβόλαια στη Solidity είναι παρόμοια με τις κλάσεις σε αντικειμενοστρεφείς γλώσσες προγραμματισμού. Ένα έξυπνο συμβόλαιο μπορεί να περιέχει μεταξύ άλλων δηλώσεις μεταβλητών κατάστασης και συναρτήσεις. Επιπλέον, τα έξυπνα συμβόλαια μπορούν να κληρονομούν από άλλα έξυπνα συμβόλαια.

Μεταβλητές Κατάστασης

Οι μεταβλητές κατάστασης είναι μεταβλητές των οποίων οι τιμές αποθηκεύονται στον μόνιμο αποθηκευτικό χώρο ενός έξυπνου συμβολαίου, δηλαδή στο blockchain, και τα δεδομένα τους παραμένουν διαθέσιμα καθ' όλη τη διάρκεια ζωής του έξυπνου συμβολαίου.

```
contract C {
  int data;  // state variable declaration
  // ...
}
```

Συναρτήσεις

Οι συναρτήσεις είναι οι εκτελέσιμες μονάδες κώδικα μέσα σε ένα έξυπνο συμβόλαιο. Δηλώνονται με τη λέξη-κλειδί `function` και μπορούν να δέχονται παραμέτρους, να επιστρέφουν πολλαπλές τιμές, καθώς και να ορίζουν ορατότητα και μεταβλητότητα κατάστασης.

```
contract C {
  function add(int a, int b) public pure returns (int sum) {
    return a + b;
  }
}
```

1.2.2 Τύποι

Οι τύποι δεδομένων της Solidity χωρίζονται σε δύο κύριες κατηγορίες: τύποι τιμής και τύποι αναφοράς.

Τύποι Τιμής

Οι τύποι τιμής κρατούν τα δεδομένα τους απευθείας στη θέση αποθήκευσής τους. Κατά τις αναθέσεις δημιουργείται μόνο ένα αντίγραφο των δεδομένων. Οι τύποι τιμής περιλαμβάνουν:

- **Τύποι integer:** Αναπαριστούν ακέραιους αριθμούς, με πρόσημο ή χωρίς, με διάφορα μεγέθη bit από 8 έως 256 σε βήματα των 8 bit, δηλωμένα ως `int8`, `int16`, ..., `int256` και `uint8`, `uint16`, ..., `uint256`. Οι τύποι `int` και `uint` είναι συνώνυμα των `int256` και `uint256`, αντίστοιχα.
- **Τύποι fixed-size byte arrays:** Αναπαριστούν ακατέργαστα δυαδικά δεδομένα σταθερού μήκους. Δηλώνονται ως `bytes1` έως `bytes32`.
- **Τύπος boolean:** Απλός τύπος που μπορεί να είναι `true` ή `false`.
- **Τύποι address:** Κρατούν μια διεύθυνση Ethereum 20 byte. Υπάρχουν δύο παραλλαγές: `address`, ο βασικός τύπος διεύθυνσης, και `address payable`, που μπορεί επιπλέον να λάβει Ether μέσω των `transfer` και `send`.
- **Τύποι enum:** Τύποι οριζόμενοι από τον χρήστη που ορίζουν πεπερασμένο σύνολο από σταθερές τιμές, παρόμοια με τις απαριθμήσεις σε C ή Java.
- **Τύποι contract:** Αντιπροσωπεύουν στιγμιότυπα έξυπνων συμβολαίων. Εννοιολογικά μοιάζουν με τύπους κλάσεων σε αντικειμενοστρεφή προγραμματισμό. Κάθε έξυπνο συμβόλαιο ορίζει τον δικό του τύπο.

- **Τύποι συναρτήσεων:** Αντιπροσωπεύουν την υπογραφή μίας συνάρτησης, συμπεριλαμβανομένων των τύπων παραμέτρων και επιστροφής, του τρόπου κλήσης (εσωτερική ή εξωτερική) και της μεταβλητότητας κατάστασης. Σημειώνονται ως:

  `function (<parameter types>) [internal | external] [pure | view | payable] [returns (<return types>)]`

  Οι τύποι συναρτήσεων μπορούν να λαμβάνουν και να επιστρέφουν πολλαπλές τιμές. Ο τρόπος κλήσης καθορίζει πώς γίνεται η κλήση στο EVM (βλ. Ενότητα 4.3.2). Επιπλέον, οι τύποι συναρτήσεων έχουν πάντα μεταβλητότητα κατάστασης, που ορίζει πως η συνάρτηση αλληλεπιδρά με την κατάσταση του έξυπνου συμβολαίου (βλ. Ενότητα 1.2.4). Οι δυνατές τιμές είναι `pure`, `view`, `payable` ή *non-payable*—η προεπιλογή όταν δεν δίνεται προσδιοριστικό.

Τύποι Αναφοράς

Οι τύποι αναφοράς αποθηκεύουν μια αναφορά (δείκτη) στα πραγματικά δεδομένα. Στη Solidity, οι τύποι αναφοράς πρέπει πάντα να συνοδεύονται από συγκεκριμένη περιοχή δεδομένων, η οποία ορίζει πού βρίσκονται φυσικά τα δεδομένα. Οι περιοχές δεδομένων αντιστοιχούν άμεσα σε διακριτές περιοχές μνήμης που διαχειρίζεται το EVM. Η Solidity παρέχει τρεις βασικές περιοχές δεδομένων: `storage`, `memory` και `calldata`:

- `storage`: Αναφέρεται στην επίμονη, επιπέδου έξυπνου συμβολαίου αποθήκευση στο blockchain. Τα δεδομένα στη `storage` γράφονται στη συνολική κατάσταση του Ethereum και παραμένουν για όλη τη διάρκεια ζωής του έξυπνου συμβολαίου. Συνεπώς, επιβιώνουν μεταξύ συναλλαγών και εξωτερικών κλήσεων συναρτήσεων.
- `memory`: Μια προσωρινή, μη επίμονη περιοχή δεδομένων που χρησιμοποιείται κατά την εκτέλεση συναρτήσεων. Η διάρκεια ζωής της `memory` συνδέεται με το τρέχον περιβάλλον εκτέλεσης (βλ. Ενότητα 4.3.2), το οποίο τυπικά ξεκινά στην είσοδο μιας εξωτερικής κλήσης συνάρτησης και λήγει όταν αυτή ολοκληρώνεται.
- `calldata`: Μη τροποποιήσιμη, μη επίμονη περιοχή δεδομένων που διαχειρίζεται αποκλειστικά το EVM ως δεδομένα εισόδου για εξωτερικές κλήσεις συναρτήσεων—ο κώδικας του χρήστη δεν μπορεί να την τροποποιήσει ούτε να δεσμεύσει δεδομένα χειροκίνητα σε αυτήν.

Επειδή οι τύποι αναφοράς λειτουργούν μέσω έμμεσης πρόσβασης, οι αναθέσεις μεταξύ τους συμπεριφέρονται διαφορετικά ανάλογα με τις περιοχές δεδομένων πηγής και προορισμού. Οι τύποι αναφοράς περιλαμβάνουν:

- **Πίνακες (Arrays):** Διακρίνονται σε κανονικούς και σε ειδικούς πίνακες `bytes` και `string`. Οι κανονικοί πίνακες μπορεί να είναι στατικού μεγέθους (`T[N]`) ή δυναμικού (`T[]`). Οι ειδικοί πίνακες αντιμετωπίζονται ως δυναμικοί πίνακες από τιμές `bytes1`, με το `string` να προορίζεται ειδικά για κείμενο κωδικοποιημένο σε UTF-8.
- **Δομές (Structs):** Σύνθετοι τύποι που ομαδοποιούν πολλαπλά πεδία διαφορετικών τύπων, παρόμοια με τις δομές σε C ή άλλες γλώσσες.
- **Αντιστοιχίσεις (Mappings):** Συσχετίζουν κλειδιά με τιμές, όπως οι αντιστοιχίες

ή τα λεξικά σε άλλες γλώσσες. Σε αντίθεση με συνήθεις δομές, δεν αποθηκεύουν τα κλειδιά άμεσα· αντίθετα, ένα κλειδί κατακερματίζεται για να προσδιοριστεί η θέση μνήμης αποθήκευσης της τιμής. Ως εκ τούτου, οι αντιστοιχίσεις δεν έχουν μήκος και δεν υποστηρίζουν επανάληψη ή απαρίθμηση κλειδιών και για αυτούς τους λόγους δεν μπορούν να αντιγραφούν. Οι αντιστοιχίσεις μπορούν να ζουν μόνο σε `storage`: βασίζονται στη δεικτοδότηση μίας αυθαίρετης θέσης σε έναν τεράστιο, αραιό χώρο από λέξεις των 32 byte, όπου οποιαδήποτε θέση μπορεί να αναγνωσθεί ή να γραφτεί άμεσα. Μη ορισμένα κλειδιά αναγιγνώσκονται ως η προεπιλεγμένη τιμή της τιμής της αντιστοίχισης. Αντιθέτως, η `memory` είναι συνεχόμενη περιοχή που αυξάνεται μόνο από το 0 προς τα πάνω· μια θέση βάσει κατακερματισμού μπορεί να βρίσκεται σε αστρονομική μετατόπιση που η `memory` δεν μπορεί να αναπαραστήσει ή να δεσμεύσει. Ο μόνος τρόπος να τροποποιηθεί μία αντιστοίχιση είναι μέσω ανάθεσης τιμών σε μεμονωμένα κλειδιά. Οι αντιστοιχίσεις ορίζονται ως `mapping(K => V)`, όπου `K` είναι ο τύπος κλειδιού και `V` ο τύπος τιμής.

1.2.3 Ορατότητα Συναρτήσεων

Η ορατότητα συναρτήσεων καθορίζει από πού μπορούν να προσπελαστούν οι συναρτήσεις και συνδέεται άμεσα με τον τρόπο κλήσης τους: εσωτερικά, εξωτερικά ή και τα δύο. Η Solidity παρέχει τέσσερα επίπεδα ορατότητας για συναρτήσεις:

- `private`: Προσπελάσιμη μόνο εσωτερικά, μέσα στο έξυπνο συμβόλαιο όπου ορίζεται.
- `internal`: Προσπελάσιμη εσωτερικά, μέσα στο έξυπνο συμβόλαιο και σε κάθε έξυπνο συμβόλαιο που κληρονομεί από αυτό.
- `public`: Προσπελάσιμη τόσο εσωτερικά (στο έξυπνο συμβόλαιο και στα έξυπνα συμβόλαια που κληρονομούν από αυτό) όσο και εξωτερικά, από οποιοδήποτε έξυπνο συμβόλαιο.
- `external`: Προσπελάσιμη μόνο εξωτερικά, από οποιοδήποτε έξυπνο συμβόλαιο.

Ο Πίνακας 1.1 συνοψίζει την προσπελασιμότητα κάθε επιπέδου ορατότητας συνάρτησης, ανά τύπο πρόσβασης και σχέσης μεταξύ έξυπνων συμβολαίων.

Visibility	Internal Access			External Access
	Same Contract	Derived Contracts	Other Contracts	Any Contract
`private`	yes	no	no	no
`internal`	yes	yes	no	no
`public`	yes	yes	no	yes
`external`	no	no	no	yes

Πίνακας 1.1: Κανόνες προσβασιμότητας για ορατότητα συναρτήσεων στην Solidity

1.2.4 Μεταβλητότητα Κατάστασης

Η μεταβλητότητα κατάστασης στη Solidity ορίζει τον τρόπο με τον οποίο μια συνάρτηση αλληλεπιδρά με την επίμονη κατάσταση ενός έξυπνου συμβολαίου. Υποδεικνύει αν μια συνάρτηση μπορεί να διαβάσει ή/και να τροποποιήσει το blockchain καθώς και αν μπορεί να λάβει Ether. Κάθε συνάρτηση σχετίζεται με μία μεταβλητότητα κατάστασης και οι περιορισμοί επιβάλλονται αυστηρά στον χρόνο μεταγλώττισης. Η Solidity ορίζει τέσσερις τύπους μεταβλητότητας κατάστασης:

- `pure`: Οι συναρτήσεις με προσδιοριστικό `pure` δεν μπορούν να διαβάσουν ούτε να τροποποιήσουν την κατάσταση του έξυπνου συμβολαίου. Αυτό σημαίνει ότι δεν έχουν πρόσβαση σε μεταβλητές κατάστασης ούτε μπορούν να καλέσουν συναρτήσεις που δεν είναι επίσης `pure`. Συνήθως χρησιμοποιούνται για υπολογισμούς που εξαρτώνται αποκλειστικά από τις παραμέτρους εισόδου.
- `view`: Οι συναρτήσεις με προσδιοριστικό `view` επιτρέπεται να διαβάζουν την κατάσταση του έξυπνου συμβολαίου, αλλά δεν μπορούν να την τροποποιούν. Μπορούν να έχουν πρόσβαση σε μεταβλητές κατάστασης και να καλούν άλλες `view` ή `pure` συναρτήσεις, αλλά δεν μεταβάλλουν το blockchain. Συνήθως χρησιμοποιούνται για επιστροφή πληροφοριών σχετικά με την τρέχουσα κατάσταση του έξυπνου συμβολαίου.
- `payable`: Μία `payable` συνάρτηση μπορεί να διαβάζει και να τροποποιεί την κατάσταση του έξυπνου συμβολαίου και επιπλέον επιτρέπεται να λαμβάνει Ether. Μόνο συναρτήσεις με προσδιοριστικό `payable` μπορούν να αποτελέσουν στόχο συναλλαγών που στέλνουν Ether. Η συγκεκριμένη μεταβλητότητα κατάστασης είναι κρίσιμη για υλοποιήσεις όπως αγορές tokens ή καταθέσεις.
- *non-payable* (έμμεσα): Οι συναρτήσεις που δεν φέρουν ρητά ένα από τα προσδιοριστικά `pure`, `view` ή `payable` θεωρούνται εξ ορισμού *non-payable*. Μπορούν να διαβάζουν και να τροποποιούν την κατάσταση, αλλά δεν μπορούν να λαμβάνουν Ether. Κάθε προσπάθεια αποστολής Ether σε τέτοια συνάρτηση θα προκαλέσει την αναίρεση της συναλλαγής.

Κάθε μεταβλητότητα κατάστασης περιορίζει τη συμπεριφορά της συνάρτησης με διαφορετικό βαθμό ελευθερίας, και ο μεταγλωττιστής διασφαλίζει ότι οι κανόνες αυτοί τηρούνται. Αυτό επιτρέπει στους προγραμματιστές να κάνουν τεκμηριωμένες υποθέσεις σχετικά με τις παρενέργειες μιας συνάρτησης.

1.3 SolGen

1.3.1 Επισκόπηση

Το SolGen είναι μία γεννήτρια τυχαίων σημασιολογικά ορθών προγραμμάτων Solidity. Αναπτύχθηκε με έμφαση στην υποστήριξη πολλών εκδόσεων του μεταγλωττιστή της Solidity. Η παλαιότερη υποστηριζόμενη έκδοση είναι η Solidity 0.5.0 και η νεότερη, κατά τον χρόνο συγγραφής, είναι η Solidity 0.8.30, αν και πολλές από τις πιο πρόσφατες δομές της γλώσσας προς το παρόν δεν έχουν υλοποιηθεί. Ο κώδικας είναι διαθέσιμος στο https://github.com/alex2449/SolGen.

Το SolGen εστιάζει αποκλειστικά στη στατική σημασιολογία των προγραμμάτων Solidity—δηλαδή στην ορθότητα όπως αυτή καθορίζεται στον χρόνο μεταγλώττισης. Διασφαλίζει ότι τα παραγόμενα προγράμματα είναι ορθά ως προς τους κανόνες ελέγχου τύπων, εμβέλειας και άλλων κανόνων της γλώσσας που ελέγχονται στον χρόνο μεταγλώττισης. Ωστόσο, δεν επιχειρεί να εγγυηθεί ότι τα προγράμματα είναι απαλλαγμένα από αόριστη συμπεριφορά (undefined behavior) και ανεξάρτητα από αδιευκρίνιστη συμπεριφορά (unspecified behavior) κατά τον χρόνο εκτέλεσης. Αυτό είναι μια σύνθετη εργασία που δεν εξετάστηκε για τους σκοπούς της παρούσας διπλωματικής. Η επιλογή αυτή απλοποιεί την υλοποίηση αλλά περιορίζει και τις δυνατότητες εντοπισμού σφαλμάτων—τουλάχιστον στα τμήματα του μεταγλωττιστή που παράγουν και βελτιστοποιούν κώδικα. Συνεπώς, η μέθοδος του differential testing δεν είναι εφικτή με το SolGen. Αντί αυτού, το κριτήριο ελέγχου (test oracle) λύνεται ελέγχοντας την απόκριση του μεταγλωττιστή: αναμένουμε ο solc να μεταγλωττίσει επιτυχώς ένα τυχαία παραγόμενο πρόγραμμα. Αν ο μεταγλωττιστής τερματίσει απρόσμενα ή αναφέρει σφάλμα—υποθέτοντας ότι η γεννήτρια δεν έχει σφάλματα—τότε υπάρχει σφάλμα στον μεταγλωττιστή, το οποίο μπορούμε να εντοπίσουμε χειροκίνητα.

Η γενική ιδέα μπορεί να περιγραφεί ως η αντίστροφη διαδικασία από αυτήν που ακολουθεί συνήθως το εμπρόσθιο τμήμα ενός μεταγλωττιστή. Αντί να πραγματοποιούμε συντακτική ανάλυση από τον πηγαίο κώδικα προς ένα αφηρημένο συντακτικό δέντρο (ΑΣΔ), κατασκευάζουμε πρώτα το ΑΣΔ και στη συνέχεια το μετατρέπουμε πίσω σε κώδικα Solidity, μετατρέποντας αναδρομικά κάθε κόμβο στην αντίστοιχη συντακτική του μορφή, σύμφωνα με τη γραμματική της γλώσσας. Καθ' όλη τη διαδικασία παραγωγής, όλες οι αποφάσεις που απαιτούν τυχαιότητα λαμβάνονται με ρίψη μεροληπτικού νομίσματος ή με επιλογή με βάρη από μια λίστα υποψήφιων στοιχείων. Τα υπόλοιπα της παρούσας ενότητας περιγράφουν τη διαδικασία παραγωγής.

1.3.2 Προσέγγιση Παραγωγής για Τύπους, Εκφράσεις και Εντολές

Το κύριο μέρος της τυχαίας παραγωγής αφορά τρία βασικά δομικά στοιχεία των περισσότερων γλωσσών προγραμματισμού:

- Τύπους, όπως `int256`, `mapping(bytes17 => string[42])`, ή `address[][10]`.
- Εκφράσεις, όπως κλήσεις ή δυαδικές εκφράσεις. Μια έκφραση αποτιμάται σε μια τιμή ορισμένου τύπου.
- Εντολές, όπως εντολές *block*, *if-else* ή *return*.

Για την παραγωγή αυτών των τριών δομικών στοιχείων, υλοποιούμε τρεις κύριες συναρτήσεις:

- `generate_type()`: Παράγει έναν τυχαίο τύπο. Η συνάρτηση χρησιμοποιείται σε ποικίλα σημεία.
- `generate_expression(type)`: Παράγει μια τυχαία έκφραση του δοθέντος τύπου. Η συνάρτηση χρησιμοποιείται για κάθε συντακτικό κανόνα—ή ισοδύναμα, για κάθε κόμβο του ΑΣΔ—που περιέχει μία έκφραση.
- `generate_statement()`: Παράγει μία τυχαία εντολή. Αντίστοιχα, η συνάρτηση χρησιμοποιείται για κάθε κόμβο του ΑΣΔ που περιέχει μία εντολή.

Καθεμία από αυτές τις συναρτήσεις αναθέτει σε εξειδικευμένες ρουτίνες την παραγωγή συγκεκριμένων υποκατηγοριών. Για παράδειγμα, η `generate_block_stmt()` παράγει εντολές block, ενώ η `generate_binary_expr(type)` παράγει δυαδικές εκφράσεις. Οι εξειδικευμένες αυτές συναρτήσεις χρησιμοποιούνται εσωτερικά από τις κύριες συναρτήσεις παραγωγής, με τον ακόλουθο τρόπο:

1. **Αρχικοποίηση υποψηφίων**: Συντίθεται μια λίστα με όλες τις παραγωγές (π.χ. τύπων, εκφράσεων ή εντολών) ως αρχικοί υποψήφιοι.
2. **Απόπειρα παραγωγής**: Επιλέγεται τυχαία μία υποψήφια παραγωγή και καλείται η αντίστοιχη μέθοδος παραγωγής.
3. **Επικύρωση**: Αν η επιλεγμένη παραγωγή δεν είναι συντακτικά ή σημασιολογικά έγκυρη στο τρέχον περιβάλλον, επιστρέφεται σφάλμα.
4. **Επανάληψη**: Η μη έγκυρη υποψήφια παραγωγή αφαιρείται από τη λίστα και επιλέγεται άλλη.
5. **Επιτυχία**: Μόλις παραχθεί έγκυρη παραγωγή, επιστρέφεται ως αποτέλεσμα.

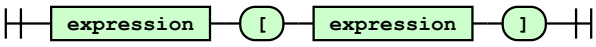
1.3.3 Παραγωγή Εκφράσεων

Ένα μεγάλο μέρος της διαδικασίας παραγωγής αφορά την παραγωγή ενός ορθά τυποδοτημένου προγράμματος που γίνεται αποδεκτό από τον ελεγκτή τύπων. Η παραγωγή εκφράσεων καθοδηγείται κυρίως από τον αναμενόμενο τύπο αποτελέσματος και από περιορισμούς που εξαρτώνται από το περιβάλλον, όπως το αν η έκφραση χρησιμοποιείται ως l-value (κάτι στο οποίο μπορούμε να αναθέσουμε).

Παραγωγή Έκφρασης Index Access

Το συντακτικό μιας έκφρασης index access δίνεται από:

κανόνας index-access-expr



Η παραγωγή της συγκεκριμένης έκφρασης περιλαμβάνει την παραγωγή δύο υποεκφράσεων: της βάσης και της θέσης, γεγονός που απαιτεί τη δημιουργία δύο τύπων, ενός για καθεμία από τις υποεκφράσεις. Εξετάζουμε τις ακόλουθες περιπτώσεις για τον τύπο της βάσης, για τις οποίες η έκφραση είναι έγκυρη:

- Αντιστοιχίσεις
- Πίνακες (κανονικοί και `bytes`)
- Πίνακες fixed-size bytes (`bytesN`)

Οι περιπτώσεις επιλέγονται τυχαία μέχρι να επιτύχει μία ή να εξαντληθούν όλες. Η καθεμία περιγράφεται ξεχωριστά παρακάτω.

Αντιστοιχίσεις

Περιορισμοί.

- Ο τύπος της έκφρασης είναι ο τύπος τιμής της αντιστοίχισης.
- Η έκφραση δεν είναι l-value αν μια ανάθεση σε αυτήν θα απαιτούσε την αντιγραφή μίας αντιστοίχισης, κάτι που δεν επιτρέπεται στη Solidity (βλ. Ενότητα 4.3.2).
- Η έκφραση δεν μπορεί να χρησιμοποιηθεί αν το περιβάλλον είναι μία `pure` συνάρτηση. Αυτό συμβαίνει επειδή οι αντιστοιχίσεις βρίσκονται πάντα στη `storage`.
- Η έκφραση δεν μπορεί να χρησιμοποιηθεί ως l-value αν το περιβάλλον είναι `pure` ή `view` συνάρτηση, για τον ίδιο λόγο.
- Ο τύπος της θέσης πρέπει να είναι αυτόματα μετατρέψιμος στον τύπο κλειδιού της αντιστοίχισης.

Διαδικασία Παραγωγής.

- Αν ο ζητούμενος τύπος δεν είναι έγκυρος τύπος τιμής αντιστοίχισης, η παραγωγή αποτυγχάνει. Έγκυροι τύποι τιμής αντιστοίχισης περιλαμβάνουν:
  - Τύπους τιμής
  - Τύπους αναφοράς που είναι δεσμευμένες `storage` αναφορές (βλ. Ενότητα 5.3.1)
  - Αντιστοιχίσεις
- Αν ζητείται l-value:
  - Αν ο ζητούμενος τύπος είναι αντιστοίχιση, η παραγωγή αποτυγχάνει (θα απαιτούσε l-value τύπου αντιστοίχισης της μορφής $\texttt{e}_1\texttt{[e}_2\texttt{]}$, κάτι αδύνατο).
  - Αν ο ζητούμενος τύπος είναι δεσμευμένη `storage` αναφορά που περιέχει αντιστοίχιση, αυτό δεν θα έπρεπε να συμβεί και υποδηλώνει σφάλμα στη λογική παραγωγής.
- Αν το περιβάλλον είναι `pure` συνάρτηση, η παραγωγή αποτυγχάνει.
- Αν το περιβάλλον είναι `view` συνάρτηση και ζητείται l-value, η παραγωγή αποτυγχάνει.
- Κατασκευή τύπου βάσης:
  - Ο τύπος κλειδιού της αντιστοίχισης παράγεται τυχαία με τη `generate_type`.
  - Ο τύπος τιμής της αντιστοίχισης ορίζεται στον ζητούμενο τύπο.
- Κατασκευή τύπου θέσης: παράγεται τυχαία ως τύπος που μετατρέπεται αυτόματα στον τύπο κλειδιού της αντιστοίχισης (βλ. Ενότητα 5.4.3).

Πίνακες

Περιορισμοί.

- Ο τύπος της έκφρασης είναι:
  - Το στοιχείο του πίνακα για κανονικούς πίνακες.
  - `bytes1` για τον ειδικό πίνακα `bytes`.
- Η έκφραση δεν είναι l-value αν:
  - Η ανάθεση σε αυτή θα απαιτούσε αντιγραφή αντιστοίχισης.
  - Η περιοχή δεδομένων του πίνακα είναι `calldata`, η οποία είναι μη τροποποιήσιμη.
- Η έκφραση δεν μπορεί να χρησιμοποιηθεί αν η περιοχή δεδομένων του πίνακα είναι

`storage` και το περιβάλλον είναι `pure` συνάρτηση.
- Η έκφραση δεν μπορεί να χρησιμοποιηθεί ως l-value αν η περιοχή δεδομένων του πίνακα είναι `storage` και το περιβάλλον είναι `pure` ή `view` συνάρτηση.
- Ο τύπος της θέσης πρέπει να είναι αυτόματα μετατρέψιμος στον τύπο `uint256`.

Διαδικασία Παραγωγής.
- Αν ο ζητούμενος τύπος δεν είναι έγκυρος τύπος στοιχείου πίνακα ή `bytes1`, η παραγωγή αποτυγχάνει. Οι τύποι στοιχείου πίνακα περιλαμβάνουν:
  - Τύπους τιμής
  - Τύπους αναφοράς που δεν είναι επαναδεσμεύσιμοι `storage` δείκτες
  - Αντιστοιχίσεις
- Αν ζητείται l-value:
  - Αν ο ζητούμενος τύπος είναι αντιστοίχιση, η παραγωγή αποτυγχάνει (θα απαιτούσε l-value τύπου αντιστοίχισης της μορφής $e_1[e_2]$).
  - Δεν θα πρέπει να είναι δυνατό ο ζητούμενος τύπος να είναι δεσμευμένη αναφορά σε `storage` που περιέχει αντιστοίχιση.
- Κατασκευή τύπου βάσης:
  - Επιλογή είδους πίνακα, επιλέγοντας μεταξύ:
    - Κανονικών πινάκων, που επιτρέπονται πάντα.
    - `bytes`, που επιτρέπεται αν ο ζητούμενος τύπος είναι `bytes1`.
  - Επιλογή περιοχής δεδομένων:
    - Το σύνολο υποψήφιων περιοχών δεδομένων είναι αρχικό κενό.
    - Αν επιλέχθηκε κανονικός πίνακας και ο ζητούμενος τύπος είναι τύπος αναφοράς ή αντιστοίχιση, τότε προσθέτουμε την περιοχή δεδομένων του ζητούμενου τύπου ή τη `storage` αντίστοιχα, επειδή οι σύνθετοι τύποι πρέπει να βρίσκονται πλήρως σε μία μόνο περιοχή δεδομένων.
    - Διαφορετικά, προσθέτουμε και τις τρεις περιοχές (`memory`, `storage`, `calldata`).
    - Εφαρμόζουμε τους εξής περιορισμούς:
      - Αφαιρούμε τη `calldata` αν ζητείται l-value.
      - Αφαιρούμε τη `storage` αν το περιβάλλον είναι `pure` συνάρτηση, ή `view` συνάρτηση και ζητείται l-value.
    - Αν το σύνολο των περιοχών είναι κενό, η παραγωγή αποτυγχάνει.
    - Διαφορετικά, επιλέγεται τυχαία μία περιοχή. Αν είναι `storage`, επιλέγεται είτε δεσμευμένη αναφορά είτε επαναδεσμεύσιμος δείκτης.
  - Αν επιλέχθηκε κανονικός πίνακας, ο τύπος στοιχείου τίθεται στον ζητούμενο τύπο.
  - Αν επιλέχθηκε κανονικός πίνακας, γίνεται τυχαία επιλογή μεταξύ στατικού ή δυναμικού. Αν είναι στατικός, επιλέγεται τυχαίο θετικό μήκος πίνακα.
- Κατασκευή τύπου θέσης: παράγεται τυχαία ως τύπος που είναι αυτόματα μετατρέψιμος στον τύπο `uint256`.

Πίνακες fixed-size bytes (`bytesN`)

Περιορισμοί.

- Ο τύπος της έκφρασης είναι `bytes1`.
- Η έκφραση δεν είναι l-value.
- Ο τύπος της θέσης πρέπει να είναι αυτόματα μετατρέψιμος στον τύπο `uint256`.

Διαδικασία Παραγωγής.

- Αν ο ζητούμενος τύπος δεν είναι `bytes1`, η παραγωγή αποτυγχάνει.
- Αν ζητείται l-value, η παραγωγή αποτυγχάνει.
- Κατασκευή τύπου βάσης: επιλέγεται έγκυρος fixed-size bytes τύπος (`bytes1`, `bytes2`, ..., `bytes32`).
- Κατασκευή τύπου θέσης: παράγεται τυχαία ως τύπος που μετατρέπεται αυτόματα στον τύπο `uint256`.

1.3.4 Παραγωγή Δηλώσεων

Είναι πιθανό η παραγωγή ορισμένων κόμβων να μην είναι άμεσα εφικτή. Για παράδειγμα, αν επιλεγεί μια έκφραση *identifier*—δηλαδή μια έκφραση που αναφέρεται σε κάποια δήλωση—για την παραγωγή έκφρασης κάποιου τύπου, μπορεί να μην υπάρχει κατάλληλη δήλωση ή να μην είναι ορατή στο τρέχον σημείο παραγωγής. Σε εκείνο το σημείο υπάρχουν δύο στρατηγικές:

- **Οπισθοδρόμηση**: Η γεννήτρια μπορεί να εγκαταλείψει την προσπάθεια παραγωγής της *identifier* έκφρασης, να την αφαιρέσει από τη λίστα υποψήφιων εκφράσεων και να δοκιμάσει άλλη υποψήφια. Αν δεν απομένουν κατάλληλες εναλλακτικές, η παραγωγή για τον συγκεκριμένο κόμβο έκφρασης αποτυγχάνει. Η αποτυχία στη συνέχεια διαχέεται προς τα πάνω, απαιτώντας την απόρριψη και γονικών κόμβων. Κάθε αλλαγή στο περιβάλλον που έγινε κατά την παραγωγή πρέπει να αναιρεθεί.

  Εξετάστε το παράδειγμα στο Σχήμα 1.1. Η γεννήτρια προσπαθεί να παραγάγει κλήση στη συνάρτηση `f`, η οποία δέχεται δύο παραμέτρους. Το πρώτο όρισμα (`new A()`) δημιουργείται επιτυχώς, με αποτέλεσμα το συμβόλαιο `A` να καταστεί εξάρτηση του συμβολαίου `B`, καθώς τα συμβόλαια εξαρτώνται από συμβόλαια που δημιουργούν μέσω `new`. Ωστόσο, το δεύτερο όρισμα δεν μπορεί να παραχθεί διότι δεν υπάρχει διαθέσιμη έκφραση του απαιτούμενου τύπου συνάρτησης εντός εμβέλειας. Συνεπώς, ολόκληρη η κλήση συνάρτησης απορρίπτεται και το συμβόλαιο `A` δεν θεωρείται πλέον εξάρτηση του συμβολαίου `B`. Αυτό είναι σημαντικό, επειδή ο γράφος εξαρτήσεων των συμβολαίων δεν πρέπει να είναι κυκλικός.
- **Κατευθυνόμενη Παραγωγή**: Εναλλακτικά, η γεννήτρια μπορεί να εξέλθει από το τρέχον περιβάλλον παραγωγής και να παραγάγει πρώτα τις απαιτούμενες εξαρτήσεις. Αναφερόμαστε σε αυτό ως *κατευθυνόμενη παραγωγή* και είναι η στρατηγική που έχει υλοποιηθεί.

```
contract A {}

contract B {
  function f(A p1, function () internal pure p2) internal {
    // ...
  }

  function g() internal {
    f(new A(), <ERROR>);
  }
}
```

Σχήμα 1.1: Παράδειγμα αποτυχίας παραγωγής και οπισθοδρόμησης

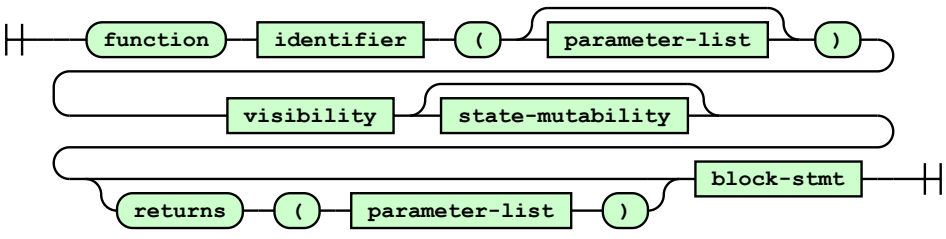
Ενώ οι δηλώσεις σε επίπεδο contract παράγονται επίσης από πάνω προς τα κάτω σύμφωνα με τη γραμματική της γλώσσας, είναι απαραίτητο να παράγονται και ως εξαρτήσεις για την παραγωγή ορισμένων εκφράσεων. Αυτό διασφαλίζει ότι η `generate_expression` θα επιτυγχάνει πάντα. Διαφορετικά, η παραγωγή έγκυρων εκφράσεων για ορισμένους πολύπλοκους τύπους θα ήταν εξαιρετικά απίθανη—ακόμη και σε μεγάλα τυχαία προγράμματα. Για παράδειγμα, θεωρήστε έναν τύπο συνάρτησης με δύο παραμέτρους και δύο τύπους επιστροφής. Αν κάθε τύπος περιορίζεται μόνο στους 64 ακέραιους τύπους της Solidity, ο συνολικός αριθμός διαφορετικών τύπων συναρτήσεων είναι περίπου 17 εκατομμύρια. Η τυχαία παραγωγή έκφρασης τέτοιου τύπου από το υπάρχον περιβάλλον είναι πρακτικά αδύνατη.

Η ενότητα αυτή καλύπτει την παραγωγή δηλώσεων επιπέδου contract ως εξαρτήσεων για την παραγωγή κάποιας έκφρασης. Υπάρχουν δύο είδη δηλώσεων που παράγονται με αυτόν τον τρόπο: ορισμοί συναρτήσεων και δηλώσεις μεταβλητών κατάστασης.

Παραγωγή Δηλώσεων Συναρτήσεων

Η αναφορά σε συνάρτηση απευθείας με το όνομά της (*κανόνας identifier-expr*) οδηγεί σε έκφραση εσωτερικού τύπου συνάρτησης. Επομένως, ένας ορισμός συνάρτησης μπορεί να παραχθεί ως εξάρτηση του κανόνα *identifier-expr* όταν:

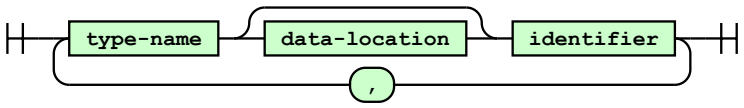
- Ο ζητούμενος τύπος είναι τύπος εσωτερικής συνάρτησης.
- Δεν ζητείται l-value, καθώς οι δηλώσεις συναρτήσεων δεν είναι αναθέσιμες.



Σχήμα 1.2: Συντακτικός κανόνας για ορισμούς συναρτήσεων

Η διαδικασία παραγωγής μίας δήλωσης συνάρτησης, όπως φαίνεται στο Σχήμα 1.2, περιλαμβάνει τα παρακάτω βήματα:

1. Παραγωγή μοναδικού αναγνωριστικού ώστε να αποφευχθούν συγκρούσεις ονομάτων.
2. Επιλογή ορατότητας (ένα από `private`, `internal`, `public`, `external`):
   - `private` και `internal` επιτρέπονται πάντα στο συγκεκριμένο περιβάλλον.
   - `public` επιτρέπεται αν κάθε τύπος παραμέτρου και τύπος επιστροφής του ζητούμενου τύπου συνάρτησης μπορεί να χρησιμοποιηθεί εξωτερικά. Ένας τύπος δεν μπορεί να χρησιμοποιηθεί εξωτερικά αν:
     - Είναι ή περιέχει εσωτερικούς τύπους συναρτήσεων.
     - Είναι τύπος αναφοράς που βρίσκεται στη `storage`.
     - Είναι αντιστοίχιση.
   - `external` δεν επιτρέπεται στο παρόν περιβάλλον, διότι οι `external` συναρτήσεις μπορούν να κληθούν μόνο εξωτερικά και δεν είναι ορατές με το όνομά τους.
3. Επιλογή μεταβλητότητας κατάστασης:
   - Αν ο ζητούμενος τύπος συνάρτησης είναι `pure`, `view` ή `payable`, προστίθεται η αντίστοιχη λέξη-κλειδί.
   - Αν ο ζητούμενος τύπος συνάρτησης είναι *non-payable*, η μεταβλητότητα κατάστασης παραλείπεται (προεπιλογή: *non-payable*).
4. Επιλογή περιέχοντος contract: το contract στο οποίο τοποθετείται η συνάρτηση επιλέγεται ως εξής:
   - Αν η επιλεγμένη ορατότητα είναι `private`, επιλέγεται το τρέχον contract.
   - Διαφορετικά, επιλέγεται οποιοδήποτε contract στη γραμμικοποίηση του τρέχοντος contract.
   - Η συνάρτηση εισάγεται σε οποιαδήποτε θέση μέσα στο σώμα του επιλεγμένου contract.
5. Κατασκευή λιστών παραμέτρων και παραμέτρων επιστροφής: κάθε τύπος παραμέτρου και τύπος επιστροφής του ζητούμενου τύπου συνάρτησης μετατρέπεται σε συντακτική δήλωση παραμέτρου (βλ. Σχήμα 1.3). Αυτό περιλαμβάνει τα εξής:
   - Κατασκευή προσδιοριστικού τύπου (type name)—συντακτική αναπαράσταση του τύπου από την εσωτερική του μορφή. Για παράδειγμα, ο τύπος `int[] memory[] memory` μετατρέπεται στο προσδιοριστικό τύπου `int[][]`. Το βήμα αυτό πρέπει να ακολουθεί την επιλογή του περιέχοντος contract (το οποίο θα χρησιμοποιηθεί ως εμβέλεια), καθώς η κατασκευή του προσδιοριστικού τύπου είναι μία διαδικασία που εξαρτάται από αυτό.
   - Επιλογή περιοχής δεδομένων:
     - Αν ο τύπος είναι τύπος τιμής, παραλείπεται η περιοχή δεδομένων.
     - Αν ο τύπος είναι τύπος αναφοράς, επιλέγεται η περιοχή δεδομένων του.
     - Αν ο τύπος είναι αντιστοίχιση, επιλέγεται `storage`.
   - Παραγωγή μοναδικού αναγνωριστικού για το όνομα της παραμέτρου ώστε να αποφεύγονται συγκρούσεις ονομάτων.
6. Παραγωγή εντολής block για το σώμα της συνάρτησης.



Σχήμα 1.3: Συντακτικός κανόνας για λίστες παραμέτρων

1.4 Πειραματικά Αποτελέσματα και Συμπεράσματα

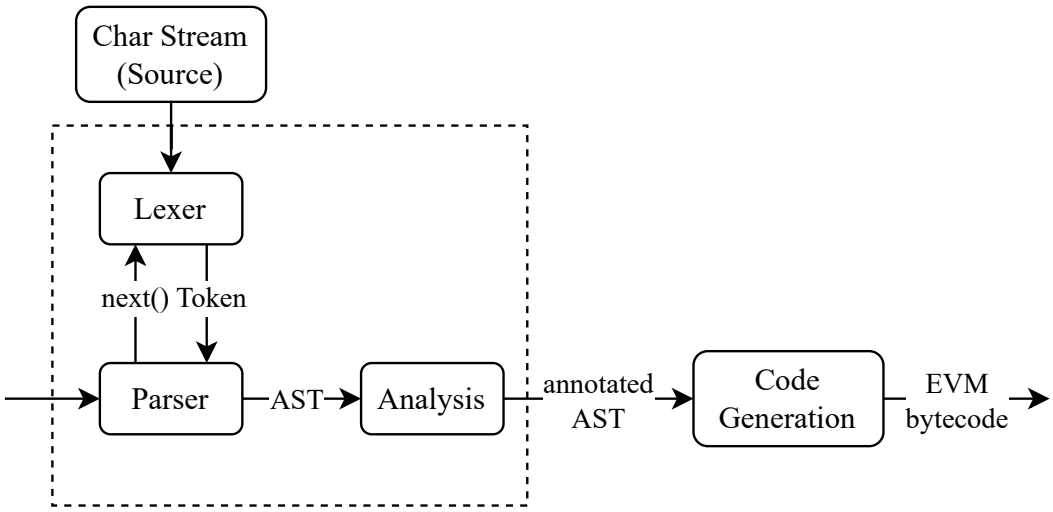
Σε αυτήν την ενότητα αξιολογούμε το SolGen ως προς την κάλυψη του κώδικα του μεταγλωττιστή της Solidity.

1.4.1 Στήσιμο

Κατασκευάσαμε τον solc 0.5.0 με ενεργοποιημένη κάλυψη κώδικα. Για τη συλλογή δεδομένων κάλυψης, μεταγλωττίσαμε 1000 τυχαία προγράμματα που παρήγαγε το SolGen χρησιμοποιώντας την προεπιλεγμένη εντολή `solc testcase.sol`, χωρίς επιπλέον ορίσματα στον μεταγλωττιστή. Εφόσον το εμπρόσθιο τμήμα εκτελείται πάντα, αυτό αρκεί για την αξιολόγηση της κάλυψής του. Το οπίσθιο τμήμα του μεταγλωττιστή, το οποίο απαιτεί ρητά ορίσματα για την εκτέλεσή του, δεν αποτέλεσε αντικείμενο της παρούσας ανάλυσης. Χρησιμοποιήσαμε lcov[1] και genhtml (το οποίο είναι μέρος του lcov) για τη δημιουργία αναφορών κάλυψης.

Το ενδιαφέρον μας επικεντρώνεται στο εμπρόσθιο τμήμα του μεταγλωττιστή, το οποίο περιλαμβάνει τη λεκτική ανάλυση, συντακτική ανάλυση και σημασιολογική ανάλυση. Το τμήμα αυτό του μεταγλωττιστή απεικονίζεται στο Σχήμα 1.4, εντός των διακεκομμένων γραμμών.

[1] https://github.com/linux-test-project/lcov



Σχήμα 1.4: Τμήματα του μεταγλωττιστή για τα οποία μετρήθηκε κάλυψη

Λεκτικός Αναλυτής

Αρχεία που αναλύθηκαν:

Από `libsolidity/parsing`: `Scanner.cpp` / `h`

	Total	Hit	Coverage
Lines	559	371	66.4%
Functions	74	60	81.1%
Branches	708	292	41.2%

Πίνακας 1.2: Κάλυψη του λεκτικού αναλυτή

Συντακτικός Αναλυτής

Αρχεία που αναλύθηκαν:

Από `libsolidity/parsing`: `Parser.cpp` / `h`, `ParserBase.cpp` / `h`

	Total	Hit	Coverage
Lines	1148	972	84.7%
Functions	143	133	93.0%
Branches	2716	992	36.5%

Πίνακας 1.3: Κάλυψη του συντακτικού αναλυτή

Σημασιολογικός Αναλυτής

Αρχεία που αναλύθηκαν:

Από `libsolidity/ast`: `ASTAnnotations.cpp` / `h`, `ASTVisitor.h`, `AST_accept.h`, `Types.cpp` / `h`

Από `libsolidity/analysis`: όλα εκτός από `DocStringAnalyser.cpp` / `h` και `SemVerHandler.cpp` / `h`.

	Total	Hit	Coverage
Lines	6312	4186	66.3%
Functions	1215	926	76.2%
Branches	14463	4103	28.4%

Πίνακας 1.4: Κάλυψη του σημασιολογικού αναλυτή

Συζήτηση

Ανάμεσα στα τρία τμήματα, ο συντακτικός αναλυτής επιτυγχάνει τη μεγαλύτερη κάλυψη, ακολουθούμενος από τον λεκτικό αναλυτή και, στη συνέχεια, τον σημασιολογικό αναλυτή.

Η λεκτική και η συντακτική κάλυψη είναι γενικά ευκολότερο να επιτευχθούν σε σχέση με τη σημασιολογική. Ένας και μόνο συντακτικός κανόνας μπορεί να έχει πολλαπλές σημασιολογικές ερμηνείες ανάλογα με τα συμφραζόμενα. Για παράδειγμα, η παραγωγή `<expr> → <identifier-expr>` μπορεί να καλύπτεται όταν γίνεται αναφορά σε μοναδική δήλωση (όπως μεταβλητής ή συνάρτησης), αλλά όχι σε περιπτώσεις υπερφορτωμένων συναρτήσεων, οι οποίες απαιτούν ξεχωριστό σημασιολογικό χειρισμό. Παρομοίως, η παραγωγή `<expr> → <expr> '=' <expr>` μπορεί να αναπαριστά διάφορα είδη αναθέσεων ανάλογα με τους τύπους των τελεστέων, καθεμία με διακριτή σημασιολογική συμπεριφορά.

Ενδιαφέρον παρουσιάζει ότι ο συντακτικός αναλυτής πετυχαίνει υψηλότερη κάλυψη από τον λεκτικό αναλυτή. Αυτό μπορεί να φαίνεται αντιδιαισθητικό, αλλά εξηγείται: η λεκτική ανάλυση περιλαμβάνει χειρισμό μη-σημασιολογικών στοιχείων όπως σχόλια, σχόλια τεκμηρίωσης (documentation comments), λεκτικές σταθερές Unicode και ακολουθίες διαφυγής. Για παράδειγμα, αν και δεν είναι δύσκολο να υλοποιηθούν, το SolGen προς το παρόν δεν παράγει παραλλαγές όπως `0x01` · εκπέμπει μόνο δεκαδικά *literals* (π.χ. `1`). Αυτές οι ελλείψεις επηρεάζουν την κάλυψη του λεκτικού αναλυτή, αλλά έχουν περιορισμένη σημασιολογική βαρύτητα.

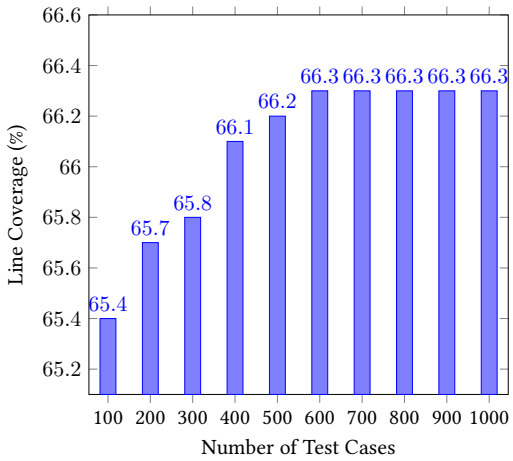
Η κάλυψη γραμμών και συναρτήσεων είναι σημαντικά υψηλότερη από την κάλυψη διακλαδώσεων. Αυτό είναι αναμενόμενο—η κάλυψη διακλαδώσεων είναι πιο απαιτητική λόγω της πολυπλοκότητας του ελέγχου ροής. Επιπλέον, το εμπρόσθιο τμήμα του μεταγλωττιστή περιλαμβάνει πολλούς κλάδους που αντιπροσωπεύουν μη έγκυρα μονοπάτια εκτέλεσης, τα

οποία το SolGen δεν επιχειρεί να παραγάγει. Τέτοιοι κλάδοι, εξ ορισμού, δεν εκτελούνται ποτέ.

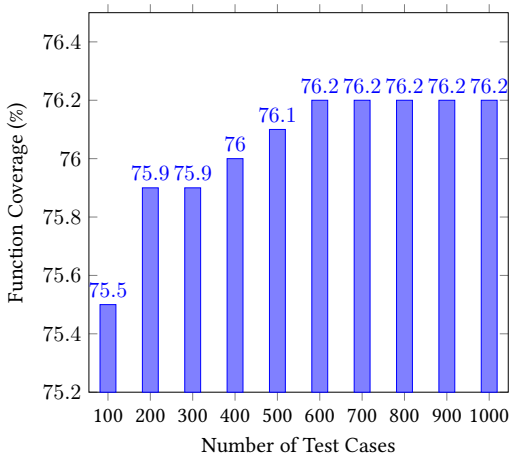
1.4.2 Σημείο Κορεσμού Κάλυψης

Για να προσδιορίσουμε πόσα τυχαία προγράμματα απαιτούνται ώστε να προσεγγιστεί η μέγιστη δυνατή κάλυψη, μετρήσαμε την αύξηση της κάλυψης για τον σημασιολογικό αναλυτή καθώς ο αριθμός των προγραμμάτων αυξανόταν από 100 σε 1000. Το συγκεκριμένο τμήμα είναι το πιο σύνθετο από τα τρία και η κάλυψή του αυξάνεται πιο σταδιακά. Αντίθετα, ο λεκτικός αναλυτής και ο συντακτικός αναλυτής φτάνουν σχετικά νωρίς τη μέγιστη κάλυψή τους—συχνά μέσα στα πρώτα λίγα εκατοντάδες προγράμματα—και επομένως εξαιρούνται από την παρούσα ανάλυση.

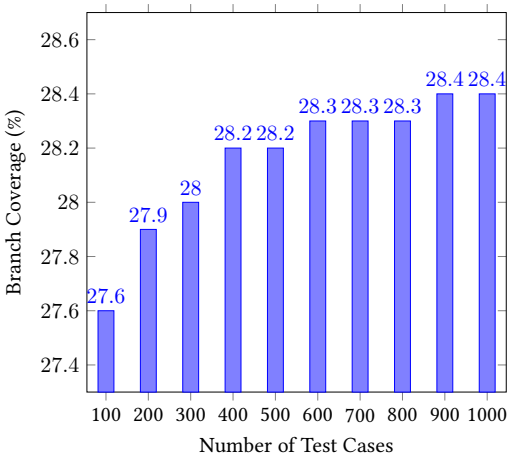
Τα Σχήματα 1.5, 1.6 και 1.7 απεικονίζουν την εξέλιξη της κάλυψης γραμμών, συναρτήσεων και διακλαδώσεων για τον σημασιολογικό αναλυτή, σε σχέση με το πλήθος των χρησιμοποιούμενων προγραμμάτων.



Σχήμα 1.5: Αυξητική κάλυψη γραμμών του σημασιολογικού αναλυτή



Σχήμα 1.6: Αυξητική κάλυψη συναρτήσεων του σημασιολογικού αναλυτή



Σχήμα 1.7: Αυξητική κάλυψη διακλαδώσεων του σημασιολογικού αναλυτή

Παρατηρούμε ότι ο ρυθμός βελτίωσης της κάλυψης μειώνεται σημαντικά μετά τα περίπου 600 προγράμματα, με μόνο μικρά οφέλη πέρα από αυτό το σημείο. Όταν φτάνουμε στα 1000 προγράμματα, τα επίπεδα κάλυψης σταθεροποιούνται περίπου στα:

- 66,3% για γραμμές,
- 76,2% για συναρτήσεις, και
- 28,4% για διακλαδώσεις.

Επομένως, θεωρούμε ότι τα 1000 τυχαία προγράμματα επαρκούν για να αποτυπωθεί η πλειονότητα της κάλυψης που είναι εφικτή με την τρέχουσα γεννήτρια.

1.4.3 Σύγκριση με τη σουίτα ελέγχου του μεταγλωττιστή

Η σουίτα ελέγχου του μεταγλωττιστή της Solidity περιλαμβάνει διάφορους ελέγχους, μεταξύ των οποίων ένα μεγάλο σύνολο συντακτικών προγραμμάτων ελέγχου. Τα προγράμματα αυτά δεν είναι σχεδιασμένα για εκτέλεση· εστιάζουν στο αν ο μεταγλωττιστής αποδέχεται ή απορρίπτει σωστά πηγαίο κώδικα με βάση τη συντακτική και σημασιολογική ορθότητά του. Σημαντικό είναι ότι περιλαμβάνουν τόσο έγκυρες όσο και μη έγκυρες περιπτώσεις. Για την ανάλυσή μας, λαμβάνουμε υπόψη μόνο τις έγκυρες—δηλαδή αυτές που ο μεταγλωττιστής αναμένεται να αποδεχτεί. Τα προγράμματα αυτά εξήχθησαν με ένα Python script και είναι συνήθως μικρά και εστιασμένα, το καθένα στοχεύοντας ένα συγκεκριμένο χαρακτηριστικό της γλώσσας, όπως μετατροπές τύπων ή πράξεις πάνω σε συγκεκριμένους τύπους.

Παραδείγματα Προγραμμάτων της Σουίτας του Μεταγλωττιστή

`implicit_conversion_from_storage_array_ref.sol`

```
contract C {
  int[10] x;
  int[] y;
  function f() public {
    y = x;
  }
}
```

`index_access_for_bytes.sol`

```
contract C {
  bytes20 x;

  function f(bytes16 b) public view {
    b[uint8(x[2])];
  }
}
```

`external_function_type_public_variable.sol`

```
contract C {
  function (uint) external public x;

  function g(uint) public {
    x = this.g;
  }
  function f() public view returns (function (uint) external) {
    return this.x();
  }
}
```

Πρέπει να σημειωθεί ότι η συγκεκριμένη σουίτα δεν αποσκοπεί στην παροχή πλήρους αυτόνομης κάλυψης, αλλά κυρίως για ελέγχους παλινδρόμησης (regression) για γνωστή συμπεριφορά.

Για να αξιολογήσουμε πώς το SolGen συμπληρώνει την υπάρχουσα σουίτα, μεταγλωττίσαμε όλα τα έγκυρα προγράμματα και εξαγάγαμε τις αντίστοιχες αναφορές κάλυψης κώδικα.

Στόχος της ακόλουθης σύγκρισης είναι να διαπιστωθεί κατά πόσο το SolGen είναι ικανό να παράγει γλωσσικές δομές που δεν καλύπτονται από τα προγράμματα ελέγχου του μεταγλωττιστή—και αντιστρόφως. Για να το κάνουμε συστηματικά, πραγματοποιούμε σύγκριση συνόλων πάνω στα δεδομένα κάλυψης.

Σύγκριση βάσει Συνόλων

Από τις αναφορές κάλυψης του lcov, εξήχθησαν τα εξής σύνολα:

- `generator_covered`: Σύνολο στοιχείων που εκτελέστηκαν από τα τυχαία προγράμματα που παρήγαγε το SolGen
- `compiler_covered`: Σύνολο στοιχείων που εκτελέστηκαν από τη σουίτα ελέγχου του μεταγλωττιστή
- `all_instrumented`: Σύνολο όλων των στοιχείων που παρακολουθούνται από τον μηχανισμό καταγραφής κάλυψης

Με βάση αυτά, ορίζουμε:

- `executed_by_both` = `generator_covered` ∩ `compiler_covered`
- `only_in_generator` = `generator_covered` \ `compiler_covered`
- `only_in_compiler` = `compiler_covered` \ `generator_covered`
- `not_covered_by_any` = `all_instrumented` − (`generator_covered` ∪ `compiler_covered`)

Το κύριο ενδιαφέρον μας εντοπίζεται στα σύνολα `only_in_generator` και `only_in_compiler`. Αυτά αντιπροσωπεύουν στοιχεία κώδικα που καλύπτονται αποκλειστικά από τη μία πηγή αλλά όχι από την άλλη και, κατά συνέπεια, μετρούν άμεσα τον βαθμό συμπληρωματικότητας μεταξύ SolGen και της υπάρχουσας σουίτας του μεταγλωττιστή. Αν το `only_in_generator` είναι μη αμελητέο, αυτό υποδηλώνει ότι το SolGen μπορεί να παράγει δομές που ενεργοποιούν τμήματα του μεταγλωττιστή τα οποία δεν αγγίζουν τα χειροποίητα προγράμματα—βασικός στόχος αυτής της αξιολόγησης.

Λεκτικός Αναλυτής

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	559	371 (66.4%)	447 (80.0%)	13 (2.3%)	89 (15.9%)
Functions	74	60 (81.1%)	66 (89.2%)	0 (0.0%)	6 (8.1%)
Branches	708	292 (41.2%)	362 (51.1%)	15 (2.1%)	85 (12.0%)

Πίνακας 1.5: Σύγκριση κάλυψης λεκτικού αναλυτή μεταξύ SolGen και σουίτας του μεταγλωττιστή

Συντακτικός Αναλυτής

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	1148	972 (84.7%)	997 (86.8%)	2 (0.2%)	27 (2.4%)
Functions	143	133 (93.0%)	136 (95.1%)	0 (0.0%)	3 (2.1%)
Branches	2716	992 (36.5%)	1022 (37.6%)	7 (0.3%)	37 (1.4%)

Πίνακας 1.6: Σύγκριση κάλυψης συντακτικού αναλυτή μεταξύ SolGen και σουίτας του μεταγλωττιστή

Σημασιολογικός Αναλυτής

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	6312	4186 (66.3%)	4860 (77.0%)	97 (1.5%)	771 (12.2%)
Functions	1215	926 (76.2%)	1009 (83.0%)	14 (1.2%)	97 (8.0%)
Branches	14463	4103 (28.4%)	5161 (35.7%)	210 (1.5%)	1286 (8.8%)

Πίνακας 1.7: Σύγκριση κάλυψης σημασιολογικού αναλυτή μεταξύ SolGen και σουίτας του μεταγλωττιστή

Η σουίτα του μεταγλωττιστή επιτυγχάνει συνολικά ευρύτερη κάλυψη (σε γραμμές, συναρτήσεις και διακλαδώσεις). Η γεννήτρια προσθέτει κάποια μοναδική κάλυψη, αλλά αυτή η αποκλειστική κάλυψη παραμένει περιορισμένη.

Αυτό αντανακλά τους τρέχοντες περιορισμούς της γεννήτριας—κυρίως λόγω ελλειπουσών δομών της γλώσσας—και όχι της τυχαίας παραγωγής προγραμμάτων γενικά. Παρότι προσφέρει κάποια συμπληρωματική αξία, τα περισσότερα μονοπάτια κώδικα έχουν ήδη ασκηθεί από τους ελέγχους του μεταγλωττιστή. Η βελτίωση της γεννήτριας θα μπορούσε να βοηθήσει στην ανάδειξη περισσότερων ακάλυπτων περιοχών.

Ωστόσο, όπως θα συζητηθεί στην ενότητα των σφαλμάτων (βλ. Ενότητα 7.2), οι μετρικές κάλυψης μπορεί να μην αποτυπώνουν πλήρως τη συνεισφορά της γεννήτριας στον εντοπισμό σφαλμάτων, όπου οι διαφοροποιήσεις στην κάλυψη μονοπατιών—η εξερεύνηση διαφορετικών σεναρίων εισόδου πάνω στα ίδια μονοπάτια κώδικα—μπορεί να παραμένουν ιδιαίτερα πολύτιμες.

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Στην παρούσα διπλωματική εργασία παρουσιάσαμε το SolGen, μια γεννήτρια τυχαίων σημασιολογικά ορθών προγραμμάτων Solidity. Αξιολογήσαμε την αποτελεσματικότητά του ως προς την κάλυψη κώδικα του μεταγλωττιστή και την ικανότητά του να εντοπίζει σφάλματα. Όπως φαίνεται στην Ενότητα 1.4, η επιτευχθείσα κάλυψη κώδικα ήταν ικανοποιητική, αλλά όχι εξαιρετική, και απαιτείται η παραγωγή προς το παρόν μη υποστηριζόμενων γλωσσικών δομών. Το SolGen εντόπισε αρκετά σφάλματα—κυρίως καταρρεύσεις μεταγλωττιστή—σε εκδόσεις του solc αρχίζοντας από την 0.5.0. Από αυτά, οκτώ ήταν μέχρι τότε άγνωστα και αναφέρθηκαν στους προγραμματιστές της Solidity.

Στόχος μας είναι να εστιάσουμε στην τελευταία διαθέσιμη έκδοση της Solidity και να επεκτείνουμε το SolGen ώστε να παράγει τις προς το παρόν μη υποστηριζόμενες γλωσσικές δομές, με σκοπό την αύξηση της κάλυψης κώδικα του μεταγλωττιστή και τον εντοπισμό μέχρι πρότινος αγνώστων σφαλμάτων. Οι κύριες γλωσσικές δομές που δεν έχουν ακόμη υλοποιηθεί είναι:

- επικάλυψη συναρτήσεων
- υπερφόρτωση συναρτήσεων
- ενσωματωμένη συμβολογλώσσα (inline assembly)
- αριθμοί και τύποι σταθερής υποδιαστολής, κυρίως επειδή ο solc δεν τους υποστηρίζει ακόμη σε μεγάλο βαθμό

Η Solidity είναι μια σχετικά νέα γλώσσα προγραμματισμού με διαρκείς αλλαγές και νέες δυνατότητες—όπως ένα ισχυρότερο σύστημα τύπων που βρίσκεται υπό ανάπτυξη—γεγονός που αφήνει σημαντικά περιθώρια βελτίωσης στη βάση κώδικα του solc. Ευελπιστούμε ότι το SolGen μπορεί να συμβάλει ουσιαστικά σε αυτήν την προσπάθεια.

Chapter 2

Introduction

2.1 Motivation

Compilers are often assumed to be bug-free and are generally trusted to function as intended. However, a compiler is itself a program and thus may contain bugs. Given the crucial role compilers play in software development, ensuring their correctness is an active and ongoing area of research. Common approaches to addressing this problem primarily involve extensive testing, including modern fuzzing and random program generation techniques [1]. In parallel, formal verification techniques—such as those used in the CompCert project—have been explored to mathematically prove compiler correctness [2].

Compilers undergo heavy testing to verify that they behave as expected. Compiler developers manually create large test suites that cover a wide range of language constructs. While this provides baseline confidence in the compiler's functionality, constructing such test suites is a tedious process, and the inherent biases of the developers may result in gaps as they cannot always predict all the unusual or edge-case uses of language constructs. This is where compiler bugs often remain hidden. This has led to efforts in random program generation, as it alleviates some of the limitations of manually created test suites.

Solidity is a programming language designed specifically for developing smart contracts that run on the Ethereum Virtual Machine (EVM) [3]. It differs from other programming languages as it must offer features and functionalities related to smart-contract-specific and EVM-specific concepts. Because Solidity smart contracts mediate real assets and execute within irreversible transactions, even subtle miscompilations can have security consequences—causing incorrect state updates, misrouted transfers, or bypassed checks. The Solidity compiler, solc, is comparatively young and evolving alongside the language and EVM, and past releases have exposed a variety of bugs. This combination of a specialized execution model and a high-stakes deployment environment makes rigorous testing especially important for establishing compiler correctness.

2.2 Aim

The aim of this thesis is to develop a generator of semantically correct Solidity programs in order to test the Solidity compiler and potentially uncover bugs. There have been other attempts at fuzzing solc, mostly by AFL-based fuzzers [4, 5, 6, 7], but to the best of our knowledge, no similar generation-based tool has been developed for Solidity. The Solidity team has developed a semantic fuzzer [8], but its goal is to test the compiler's optimizer, and it operates at the level of the language's intermediate representation, which is known as Yul.

2.3 Delimitations

Random program generation for an entire programming language is a complex and time-consuming task that exceeds the scope of this work. As a result, the generator is restricted to a subset of Solidity's features, with the unimplemented features considered for future work.

The random program generator is specifically implemented for Solidity, and the semantic restrictions are hard-coded within the internals of the generator. While a more general tool could be created, which would receive the semantic restrictions as input, this was not the focus of this work.

Lastly, the detection of unintended behavior in the compiler is limited to compile time, as emphasized in Section 5.1.

2.4 Contributions

Using SolGen, we detected bugs across different versions of the Solidity compiler, including some in the latest released versions; several were previously unknown and were reported to the Solidity developers. The primary contribution is not in the quantity of bugs found, but in providing a tool that can reduce the need for manually constructing test cases.

2.5 Thesis Structure

The remainder of this thesis is structured as follows:

In the next Chapter, we provide background on compiler testing and relevant concepts, such as fuzzing techniques, test oracles, and approaches to the test oracle problem.

In Chapter 4, we give an overview of core Solidity concepts and language features that are relevant to this thesis.

In Chapter 5, we introduce SolGen, the proposed random program generator, and examine the generation process for various language constructs.

In Chapter 6, we describe the implementation of SolGen and relevant internal components.

In Chapter 7, we evaluate SolGen using code coverage metrics and discuss bugs discovered during testing.

In Chapter 8, we conclude the thesis and outline limitations and directions for future work.

Chapter 3

Background

3.1 Compilers

Compilers are pieces of software that translate programs written in a high-level programming language (the source) into a lower-level language (the target), such as assembly or machine code, to produce an executable program. Compilers are complex and consist of several stages, with each stage transforming its input and passing its output to the next one. These stages are known as the compilation pipeline and typically consist of lexical analysis, syntactic analysis, semantic analysis, translation to an intermediate representation (IR), optimization, and machine code generation. Compilers are essential in the software development process, as every piece of software has been processed by a compiler or a compiler-like tool.

3.2 Compiler Correctness

A compiler is considered correct if it rejects invalid programs with appropriate error messages and, for valid programs, produces target code that preserves the intended behavior of the original program. Since source programs are not executed directly, their intended behavior is determined by the semantics of the source language, and the compiled program must exhibit that same behavior at runtime. Common compiler bugs include:

- Crashes, such as segmentation faults or internal compiler errors (ICEs), meaning that the compiler has been driven into an internally unexpected or inconsistent state.
- Long compilation times, which may be caused by infinite loops or performance issues.
- Incorrect diagnostics, where the compiler either fails to report an error or erroneously reports one.
- Invalid code generation: the most severe type of bug, as it can affect deployed code, often in a "silent" way that can be hard to detect. These are most likely the result of a bug in an optimization pass, since optimization passes perform the most transformations on the source code and are by far the most complex components of

the compilation pipeline.

Bugs in compilers can lead to incorrect program behavior, security vulnerabilities, software crashes, data corruption or other serious consequences. As a result, ensuring compiler correctness is crucial for the reliability and functionality of software, especially in safety-critical or security-sensitive domains. Recognizing the importance of compiler correctness, two main approaches have been adopted: formal verification and testing.

3.3 Formal Verification

Formal verification uses mathematical proofs to guarantee that a compiler's translation process is correct. It requires a formal semantics specification for both the source and target languages, as well as a specification for the translation itself. This method provides strong correctness guarantees and is especially valuable for safety-critical applications, where the correctness and reliability of software are of utmost importance. However, formally verified compilers are not typically used in general-purpose settings, as they are outperformed by production-grade optimizing compilers. The state-of-the-art in formal verification is CompCert [2], an optimizing compiler for Clight (a large subset of the C programming language), which produces code whose performance barely matches that of GCC at -O1. Additionally, the development and maintenance of formally verified compilers are extremely time-consuming and require a high level of expertise in formal methods, which limits their practical adoption. Nevertheless, formal verification remains an active area of research.

3.4 Testing

Testing involves designing and executing a set of both positive and negative test cases that cover a wide range of a language's constructs. By comparing the output of the compiler with the expected output, testing can detect any deviations or errors in the compiler's behavior. Testing alone cannot guarantee complete correctness, as it is practically impossible to exhaustively test all possible inputs. Compiler developers manually create large test suites. However, given the size and complexity of modern programming languages and compilers themselves, this is a tedious process. For example, the Microsoft Visual C++ compiler test suite consists of *hundreds of thousands* [9] of small tests. Additionally, manually written test suites often miss "weird" or uncommon edge or corner cases that may not be properly handled by the compiler. This has led to efforts in automated test case construction, which are used in a method known as *fuzz testing* or *fuzzing*.

3.5 Fuzzing

Fuzzing is the process of continuously providing random input, valid or invalid, generated by one program, referred to as the *fuzzer*, to another program we want to test, such as a compiler, and monitoring its response in an attempt to discover bugs. This also requires a criterion to determine whether a test has passed or failed (discussed in Section 3.6). Fuzzing was first used at the University of Wisconsin in the 1980s by Professor Barton Miller to test UNIX utility programs. The results were published in 1990 [10]. Since then, fuzzing has developed significantly with more sophisticated techniques [11], making it an effective and widely used technique for finding security bugs and vulnerabilities in software. Fuzzing techniques can be categorized into three categories [12]: structured or unstructured, white-box, gray-box, or black-box, and generation- or mutation-based.

3.5.1 Input Structure Awareness

We could generate completely random data as input, which in this case would be considered unstructured. This would be highly ineffective for testing programs that receive input that has some sort of structure. For example, compilers receive programs that conform to a formal grammar, servers receive packets with specific fields of a known size, etc. A program that is not syntactically correct does not get past the parsing stage of a compiler and thus does not explore deeper code in the compilation pipeline; a packet with random fields might not even reach its destination, etc. In this case, a fuzzer would need to be aware of the structured nature of the input, making the fuzzing structured.

3.5.2 Program Structure Awareness

Another aspect is whether the input construction is guided by the internal workings of the system under test or not. For example, input can be generated by what yields better code coverage, according to some instrumentation. This is known as white-box testing. However, measuring coverage is expensive, so different heuristics have been implemented to deal with this. In this case, the fuzzer is still guided towards increasing code coverage but with a reasonable performance overhead. This is known as gray-box testing. Testing without knowledge of the underlying implementation is known as black-box testing.

3.5.3 Generation- and Mutation-based Construction

There are two approaches for constructing random input [12]. Generation-based methods involve creating input from scratch. Mutation-based methods involve taking an existing input and mutating it through a series of transformations to generate a new input. Mutation-based fuzzing is generally quicker and easier to implement, but its effectiveness is dependent on the existing input and thus may result in less variety. Generation-based fuzzing, on the other hand, is more time-consuming and complex, but it can be more

effective at uncovering deep, complex bugs that are not easily triggered by simple mutations.

3.6 Test Oracles

But how can we know whether a compiler has the desired behavior for a given input program? This is known as the test oracle problem. Ideally, we would like to know what the correct output should be in order to compare it with the actual output produced by the compiler being tested. Any deviation between the two would indicate a bug. This is not possible for testing compilers (or any complex software) where the correct output is not easily defined since there is no formal specification of what that output should be—compilers are free to generate and optimize code as they see fit.

For compiler testing, tests can be divided into two categories: positive and negative. Negative test cases are easier to examine since we expect a compiler to simply reject them and exit normally. Positive test cases can be examined at compile time or run time. At compile time, crashes, internal compiler errors, timeouts or any error messages can be looked at to determine whether the compiler executed as intended. However, if the input program is compiled successfully, we don't know whether its run-time behavior is correct or a miscompilation has occurred. There are two main approaches that attempt to address this: metamorphic testing and differential testing.

3.6.1 Metamorphic Testing

Metamorphic testing is an approach to solving the test oracle problem proposed by T. Y. Chen et al. [13]. It involves finding metamorphic relations which define how the output should change in response to changes in the input. A popular case of metamorphic testing is *Equivalence Modulo Inputs* (EMI) [14], where an input is mutated in a way such that it should produce the same output. This is illustrated in Figure 3.1. A simple example of metamorphic testing under EMI is testing a sorting algorithm. Once the algorithm is implemented and tested with a specific input, metamorphic testing can be used to generate new inputs by reversing the order of elements in the input, shuffling the elements or adding duplicate elements. The algorithm should produce the same output for the new inputs as it did for the original input.

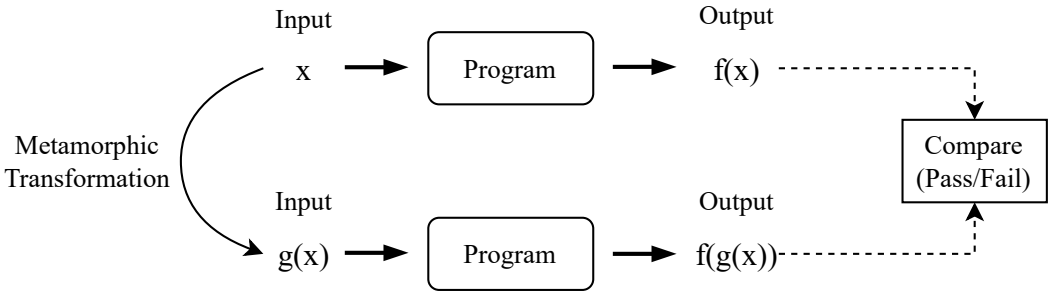


Figure 3.1: Visualization of metamorphic testing under EMI

3.6.2 Differential Testing

Differential testing was proposed by McKeeman [15] as one way to solve the test oracle problem. In the domain of compiler testing, the idea is that a program should produce the same output, regardless of how that program was compiled, assuming that it is deterministic, terminating, free of undefined behavior and independent of unspecified behavior. We can compile a program with multiple different compilers for the same language (e.g., GCC and Clang for C/C++), under different optimization settings (for finding bugs in the optimizer) or under different versions of the same compiler (for finding regression bugs) and compare execution results. Figure 3.2 illustrates differential testing for two compilers.

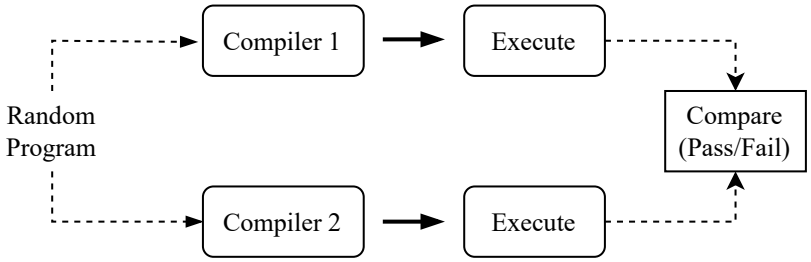


Figure 3.2: Visualization of differential testing for two compilers

3.7 Constructing Valid Programs

With both metamorphic and differential testing, it is crucial that the constructed programs have a well-defined behavior at runtime, meaning that programs must be free of undefined behavior and independent of unspecified behavior at runtime. Otherwise, a deviation between execution results wouldn't necessarily indicate a bug. For example, accessing an array out-of-bounds is considered undefined behavior in C and should be avoided. Similarly, the order of evaluation of an operator's operands or a function call's arguments is unspecified. The reason for this is to allow compilers to perform aggressive optimizations.

Csmith [16], a random program generator for C, employs different strategies for generating

programs that avoid the 191 kinds of undefined behavior and are independent of the 52 kinds of unspecified behavior in C. These are both compile-time and run-time solutions. For example, to ensure that an array index is always in-bounds, Csmith may apply the modulo operator on the array index with the array's length. In other situations, more complex strategies have to be applied. For example, in order to avoid unspecified behavior in the order of evaluation of subexpressions, Csmith performs effect analysis. As a safety mechanism, it then performs a series of checks and only submits a code fragment if it is verified to be safe.

3.8 Test Program Reduction

Test programs tend to be large and complex. Chen et al. found that for Csmith, the largest number of bugs was found in an 81KB program. Such a program is not helpful for compiler developers; they instead want reporters *"...to narrow down the bug so that the person who fixes it will be able to find the problem more easily..."* [17]. Test program reduction aims to reduce a test program to a smaller and simpler one that still triggers the same bug. Reduced test cases not only simplify bug diagnosis, but also serve as concise regression tests in future versions.

The core idea behind test program reducers such as C-Reduce [18] is Delta debugging [19]. In short, the delta debugging algorithm starts with an input and incrementally reduces it by removing or simplifying parts of it. After each reduction, it re-runs the program to check if the failure still occurs. If the failure persists, it continues the reduction process iteratively until the smallest possible input that still triggers the bug is identified.

3.9 Related Work

Randomized testing has proven effective at uncovering compiler bugs across many languages; below we outline methods ranging from byte-level fuzzing to grammar/AST-guided and semantics-aware generation.

AFL [20] (and LibFuzzer [21]) are classic greybox fuzzers: they mutate bytes, run the target, and keep inputs that increase coverage. This is simple and powerful, but for compiler inputs (programs) it hits a limit: byte-level mutations rarely preserve syntax, so most mutated files are rejected by the parser and never reach type checking or optimization.

To address this, LangFuzz [22] adds structure awareness. It turns seed programs into ASTs and replaces one node (e.g., an expression or statement) with another of the same kind so the result stays syntactically valid. This small change is crucial: valid inputs survive parsing and can reach later stages where interesting compiler bugs live.

Superion [23] advances this idea with grammar-aware trimming (keep inputs small yet valid) and targeted AST mutations that swap subtrees only when the position is

syntactically—and, where applicable, type-compatible. Because the fuzzer edits structure rather than bytes, the resulting programs typically parse and type-check, enabling exploration of semantic and optimization passes that byte-level fuzzers miss.

Le et al. [14, 24] apply metamorphic testing to validate optimizing compilers in their tools Orion and Athena. They produce variants of a program that should produce the same output for a given set of inputs. First, they profile executions of the program over the set of inputs to discover which code regions are not executed under the specific inputs. They then prune, insert or modify the program's unexecuted code. The mutated program's runtime behavior should obviously remain the same under the same set of inputs. In a period of eleven months, Orion found a total of 147 unique bugs across GCC and LLVM. Athena found 72 bugs.

Le et al. [25] further improved this idea to not only mutate dead code regions but also live code, with their tool Hermes. The idea is to select a program point and insert code such that the program state after the inserted code matches the state before its execution. This requires that during the profiling stage, the value of every live variable at the selected point is recorded. Hermes found a total of 168 bugs, which Orion and Athena missed.

Nautilus [26] combines generation and mutation under coverage guidance. It can generate programs directly from a grammar (removing reliance on a large seed corpus) and then mutate the AST guided by edge/branch coverage. The key point is that the combination of coverage-guided search with structured program representations yields a steady stream of valid, diverse test cases that reach deeper compiler logic.

Similar to our work, there are methods that make generation semantics-aware. Van Heerden et al. [27] introduce semantic mark-up tokens—lightweight annotations that encode scope/typing constraints—to make grammar-based generators produce contextually valid programs. Kifetew et al. [28] use stochastic context-free grammars with semantic annotations (e.g., types, bindings) so derivations tend toward well-typed programs, sometimes coupled with genetic programming to steer toward higher coverage. Dewey et al. [29] express grammars and typing rules as constraints and use a solver to generate programs that specifically stress components like a type checker (shown for Rust). These ideas keep programs "valid enough" to exercise compiler internals, rather than just crashing at the front end.

Csmith [16] is a successful random program generator for the C programming language that utilizes differential testing, having found 325 previously unknown bugs in mainstream C compilers. Csmith is capable of generating complex C programs that use many of the language's features. It is based on the C grammar, and it uses complex heuristics to ensure that programs are free of undefined behavior and independent of unspecified behavior.

Building on Csmith's idea, CLsmith [30] adapts the generator to OpenCL kernels and uses differential testing across many compiler/device pairs. It adds OpenCL-specific controls

(e.g., address-space qualifiers like _global/_local, kernel attributes) so kernels are valid for the OpenCL execution model. As an oracle, it employs equivalence modulo inputs (EMI): generate variants that should behave identically for a fixed input; if two variants disagree on some platform, that flags a compiler/driver bug. In short, CLsmith shows how a carefully engineered C generator can be transferred to OpenCL by respecting the new semantics and pairing it with a scalable oracle.

Finally, property-based testing, exemplified by QuickCheck [31] flips the perspective: you write a property, and the framework randomly generates inputs trying to falsify it. In compiler contexts, two patterns are useful. First, type-directed generation: treat types (or typing judgments) as generators, so inputs are well-typed by construction and failures are more likely to come from deeper passes. Second, metamorphic properties: apply semantics-preserving transformations (e.g., re-associating additions, constant folding) and check that results are equivalent—a QuickCheck-style way to get EMI-like checks without a reference compiler.

Chapter 4

Solidity

Solidity is a statically-typed, contract-oriented programming language designed specifically for developing smart contracts that execute on the Ethereum Virtual Machine (EVM). The EVM is the decentralized runtime environment responsible for executing smart contracts on the Ethereum blockchain. Unlike general-purpose programming languages, Solidity is tailored to the unique requirements of blockchain applications, offering features such as the ability to send and receive Ether (Ethereum's native cryptocurrency), transaction-specific error handling (e.g., reverting or aborting), and explicit control over distinct data storage locations within the EVM (`storage`, `memory`, and `calldata`). While a more thorough description of the language can be found in its official documentation, this chapter provides an overview of some core language constructs and behaviors that are relevant to this thesis.

4.1 Structure of a Contract

Contracts are the building blocks of a Solidity source file. Contracts in Solidity are similar to classes in object-oriented programming languages. A contract can contain declarations of state variables, functions, function modifiers, constructors, type definitions and more. Furthermore, contracts can inherit from other contracts.

State Variables

State variables are variables whose values are permanently stored in contract storage, i.e., in the blockchain, and their data persists for the lifetime of the contract.

```
1  contract C {
2    int data;  // state variable declaration
3    // ...
4  }
```

Functions

Functions are the executable units of code within a contract. They are declared using the `function` keyword and can accept parameters, return multiple values, and specify visibility and state mutability.

```
contract C {
  function add(int a, int b) public pure returns (int sum) {
    return a + b;
  }
}
```

Constructors

A constructor is a special function that is executed once when the contract is deployed. It is used to initialize contract state. Solidity does not support constructor overloading—each contract can have at most one constructor. If none is declared, the compiler automatically provides a default constructor.

```
contract C {
  int data;
  constructor(int _data) {
    data = _data;
  }
}
```

Function Modifiers

Modifiers are used to alter the behavior of functions in a declarative manner, typically for adding checks, restrictions, or logging. A modifier can wrap around a function body using the special placeholder statement `_;`, which indicates where the original function body will be inserted.

```
contract ModifierExample {
  address owner;

  constructor() {
    owner = msg.sender;
  }

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function withdraw() public onlyOwner() {  // modifier invocation
    // Only the owner can call this function.
    // Otherwise, the transaction is reverted.
    // ...
  }
}
```

Events

Events allow contracts to emit logging information that is stored on the blockchain and can be captured by off-chain applications. They are declared using the `event` keyword and emitted using the `emit` keyword.

```
contract C {
  int data;
  event DataUpdated(int newValue, uint timestamp);

  function update(int x) public {
    data = x;                              // update contract state
    emit DataUpdated(x, block.timestamp);  // emit event with timestamp
  }
}
```

4.2 Types

Solidity's data types are divided into two main categories: **value types** and **reference types**.

4.2.1 Value Types

Value types hold their data directly in their storage location. Only a copy of the data is made in assignments or when passing function call arguments. Value types include:

- **Integer types:** Represent whole numbers, either signed or unsigned and of various bit widths, ranging from 8 to 256 bits in 8-bit increments, specified as `int8`, `int16`, ..., `int256` and `uint8`, `uint16`, ..., `uint256`. The types `int` and `uint` are aliases for `int256` and `uint256`, respectively.
- **Fixed-size byte arrays:** Represent raw binary data of fixed length. These are declared as `bytes1` through `bytes32`.
- **Boolean:** A simple type that can be either `true` or `false`.
- **Address types:** Hold a 20-byte Ethereum address. There are two variations: `address`, a basic address type, and `address payable`, which can additionally receive Ether via the `transfer` and `send` functions.
- **Enum types:** User-defined types that define a finite set of constant values, similar to enumerations in languages like C or Java.
- **Contract types:** Represent the instances of contracts. They are conceptually similar to class types in object-oriented programming. Every contract defines its own type.
- **Function types:** Represent the type signature of a function, including its parameter and return types, calling convention (internal or external), and state mutability. They are notated as:

  `function (<parameter types>) [internal | external] [pure | view | payable] [returns (<return types>)]`

  Function types can receive and return multiple values. The calling convention

determines how the function is invoked in the EVM (see Section 4.3.2). Additionally, function types always have a state mutability, which defines how the function interacts with contract state (see Section 4.4.4). The possible values are `pure`, `view`, `payable`, or *non-payable*—the default when no specifier is given.

4.2.2 Reference Types

Reference types store a reference (or pointer) to the actual data. In Solidity, reference types must always be associated with a specific data location, which defines where the data physically resides. These data locations directly correspond to distinct memory regions managed by the EVM. Solidity provides three main data locations: `storage`, `memory`, and `calldata`:

- `storage`: Refers to persistent, contract-level storage on the blockchain. Data stored in `storage` is written to Ethereum's global state and remains for the entire lifetime of the contract. As such, it persists between transactions and external function calls.
- `memory`: A temporary, non-persistent data area used during function execution. The lifetime of `memory` is tied to the current execution context (see Section 4.3.2), which typically begins at the entry point of an external function call and ends when that call completes.
- `calldata`: A non-modifiable, non-persistent data area managed exclusively by the EVM as input data for external function calls, meaning user code cannot modify it or manually allocate data in it.

Because reference types work through indirection, assignments between them behave differently depending on the source and destination data location. Reference types include:

- **Arrays:** Divided into ordinary arrays and the special arrays `bytes` and `string`. Ordinary arrays can be statically sized (`T[N]`) or dynamically sized (`T[]`). Special arrays are treated as dynamic arrays of `bytes1` values, with `string` specifically intended for UTF-8 encoded text.
- **Structs:** Custom data types that group together multiple fields of varying types, similar to structs in C or other languages.
- **Mappings:** They associate keys with values, similar to maps or dictionaries in other languages. Unlike typical data structures, they do not store keys directly; instead, a key is hashed to determine the storage slot for its value. Consequently, mappings have no length and do not support iteration or key enumeration, and for these reasons they cannot be copied. Mappings can live only in `storage`: they rely on addressing an arbitrary slot in a huge, sparse space of 32-byte words, where any slot can be read or written directly. Unset keys read as the value type's default. By contrast, `memory` is a contiguous region that grows only from 0 upward; a hash-based location may lie at an astronomical offset that `memory` cannot represent or allocate. The only way to modify a mapping is by assigning values to individual keys. Mappings are specified as `mapping(K => V)`, where `K` is the key type and `V` is the value type.

4.3 Expressions

4.3.1 Assignment Semantics for Reference Types

Solidity distinguishes between value types and reference types, each with different assignment semantics. Value types store their data directly in their respective locations. As a result, assignments or function calls involving value types result in copies of the data. In contrast, reference types store a pointer to the actual data rather than the data itself. This distinction affects how assignments behave and whether they result in a data copy or merely a reference to the same underlying data. The assignment semantics for reference types can be summarized as follows:

Assignments to `storage`. Assignments to `storage` are allowed from any data location (`memory`, `calldata`, or `storage`) and always result in an independent deep copy of the data. Because the data is fully copied, the compiler allows more flexibility in the types involved, as shown in the example below:

```
contract C {
  uint16[] a = [42];
  uint32[] b;

  function f() internal returns (uint16, uint32) {
    b = a;                // copies contents of a into b
    b[0] = 17;            // modifies only b's copy, a stays intact

    return (a[0], b[0]);  // returns (42, 17)
  }
}
```

In this example:

- Assigning `a` to `b` copies the entire array data from one `storage` location to another, creating an independent copy. Note that the element type of `a` (`uint16`) is not identical to `b`'s (`uint32`), but it is implicitly convertible.
- Modifying `b[0]` after the assignment affects only `b`, leaving `a` unchanged.
- Returning both values demonstrates that `a` and `b` hold separate copies of the data. The same deep copy behavior applies when assigning from `memory` or `calldata` to `storage`.

Assignments to Local `storage` Variables. Assignments to a local `storage` variable (i.e., a `storage` pointer declared within a function) are only allowed from existing `storage` variables or other local `storage` variables. These assignments do not copy data but instead create a reference to the same storage location.

```
contract C {
  uint[] a = [42];

  function f() internal returns (uint) {
    uint[] storage p = a;  // p points to a's storage slot
    p[0] = 17;             // modifies a[0] through p

    return a[0];           // returns 17
  }
}
```

In this example:

- The local `storage` variable `p` is assigned from the state variable `a`, meaning both refer to the same `storage` location.
- Mutating `p[0]` directly affects `a[0]`.
- Returning `a[0]` confirms that the change through `p` is visible in `a`.

Assignments from other data locations are not allowed, as this would require the creation of a temporary in `storage` just to give the `storage` pointer something valid to point to, which is not supported, since `storage` is intended for persistent data and is not designed to hold temporary values.

```
contract C {
  function f(int[] memory a) public {
    int[] storage p = a;  // error
  }
}
```

Assignments to `memory`. Assignments to `memory` are allowed from any data location (`memory`, `calldata`, or `storage`). When assigning from `memory`, only a reference is copied, not the underlying data. As a result, two `memory` variables may point to the same data, and modifying one affects the other. However, when assigning from `storage` or `calldata`, an independent deep copy of the data is created.

```
contract C {
  uint8[1] a = [42];

  function f() internal view returns (uint8, uint8) {
    uint8[1] memory x = [42];  // x points to a temporary in memory
    uint8[1] memory y = x;     // y points to the same memory as x
    y[0] = 17;                 // modifies x[0] through y

    uint8[1] memory z = a;     // deep copy from storage
    z[0] = 17;                 // a[0] is unaffected

    return (x[0], a[0]);       // returns (17, 42)
  }
}
```

In this example:

- The variable `x` is initialized in `memory` and then assigned to `y`, which results in a reference assignment — both `x` and `y` point to the same `memory` region. Modifying `y[0]` also changes `x[0]`.
- The variable `z` is assigned from the storage variable `a`. This results in a deep copy, meaning `z` now holds an independent copy of the array in `memory`.
- Modifying `z[0]` does not affect the original `a[0]` in `storage`. This is also why the function can be marked as `view`. Functions marked `view` are prohibited from modifying `storage` (see Section 4.4.4).
- The return values demonstrate this distinction: `x[0]` reflects the mutation (since `y` modified it), while `a[0]` remains unchanged, confirming that a deep copy occurred.

Assignments to `calldata`. Assignments to `calldata` are highly restricted: they are only allowed from other `calldata` references and always result in a reference, never a copy. This is because data in `calldata` cannot be manually allocated by user code.

Table 4.1 summarizes the validity and copying behavior of assignments between reference types in different data locations.

Note: Assignments between different data locations always involve a deep copy, because references in one location (e.g., `memory`) cannot safely point to data in another (e.g., `storage` or `calldata`).

Destination	Source	Valid?	Copy Performed?
`storage`	`storage` / `memory` / `calldata`	yes	yes
local `storage` pointer	`storage`	yes	no
	`memory` / `calldata`	no	–
`memory`	`memory`	yes	no
	`storage` / `calldata`	yes	yes
`calldata`	`calldata`	yes	no
	`storage` / `memory`	no	–

Table 4.1: Overview of assignment semantics for reference types

4.3.2 Function Calls

At the EVM level, control flow between functions is handled using two key instructions: `CALL` and `JUMP`. These instructions differ fundamentally in how they manage the execution context.

The execution context refers to the runtime environment in which a contract function executes. It includes, among other things:

- the contract's bytecode and address,
- the contract's `storage` (its persistent state on the blockchain),
- a temporary memory region, corresponding to Solidity's `memory` data location,
- and the calldata passed to the function, corresponding to Solidity's `calldata` data location.

The two instructions behave as follows:

- The `JUMP` instruction simply updates the program counter to another instruction within the bytecode of the current execution context, meaning the callee operates in the same execution context as the caller, sharing the same `memory` and `storage`.
- The `CALL` instruction creates a new execution context, in which the callee executes in isolation from the caller, with its own separate `memory` region.

This gives rise to two kinds of function calls in Solidity: **internal** and **external**.

Internal Calls

Internal calls are compiled to the EVM's `JUMP` instruction and execute within the same execution context.

To perform an internal call, a function must be invoked directly by name, not via a contract instance. In the example below, the expression `f` evaluates to an internal function type, which causes the compiler to treat the call `f()` as an internal function call:

```
contract C {
  function f() public pure { }
  function g() public pure {
    f();  // internal call of function f
  }
}
```

External Calls

External calls are compiled to the EVM's `CALL` instruction and create a new execution context.

To perform an external call, a function must be invoked through a contract instance—the contract instance to which the message call is made. In the example below, the expression `this.f` evaluates to an external function type, which causes the compiler to treat the call `this.f()` as an external function call. The special keyword `this` refers to the current contract instance:

```
contract C {
  function f() public pure { }
  function g() public view {
    this.f();  // external call of function f
  }
}
```

Parameters for External Calls

Internal function calls can use any parameter and return type without restriction. This is because both the caller and the callee operate in the same execution context, sharing access to `memory`, `storage`, and the contract's deployed bytecode. As a result, types that depend on the current execution context can be passed safely.

External function calls, on the other hand, are handled through the Ethereum Application Binary Interface (ABI), which enforces a strict boundary between contracts. Since the caller and the callee do not share an execution context, types that depend on the current execution context are disallowed. These include:

- **Reference types in `storage`:** These point to a contract's persistent storage, which is isolated per contract in the EVM. This is a design decision at the EVM level: when accessing `storage` (via the `SLOAD` and `SSTORE` instructions), the EVM uses the contract address from the current execution context to locate the `storage` structure for that contract. As a result, `storage` references are only valid within their owning contract's execution context, and since an external call creates a new execution

context for the recipient contract, the reference becomes invalid and the Solidity compiler disallows passing or returning them in external calls.

- **Internal function types:** Internal function calls use the EVM's `JUMP` instruction, which sets the program counter to a specific byte offset within the current contract's deployed bytecode. This jump mechanism relies on the caller and callee sharing the same execution context. An internal function type is essentially a pointer to a location in the current contract's bytecode. Since external calls switch to a new execution context, internal function types become invalid, and the compiler disallows passing or returning them in external calls.

The example below shows invalid uses of types in external contexts:

```
contract C {
  int[] v;

  // error: external function returns a storage reference
  function f() external view returns (int[] storage) {
    return v;
  }

  // error: external function receives internal function pointer
  function g(function () internal pure) external pure { }
}
```

Reference Types in External Function Signatures

External calls are handled via the Ethereum ABI. The ABI specifies a standardized way to encode function arguments into a binary payload that can be sent between contracts. This encoded payload is stored in a special, read-only memory region called `calldata`, which acts as the immutable input buffer for external function invocations.

In Solidity, externally callable functions can specify the data location of reference-type parameters as either `memory` or `calldata`. However, the external function type derived from such functions always uses `memory` for reference types, even if the actual function parameters are declared as `calldata`, as demonstrated in the example below:

```
contract C {
  function f(int[] calldata x) external pure { }

  function g() internal view {
    // error: incompatible types
    function (int[] calldata) external pure fp1 = this.f;

    // this is valid
    function (int[] memory) external pure fp2 = this.f;
  }
}
```

This behavior exists because from the caller's perspective, arguments must be prepared in `memory` before ABI encoding. The ABI encoder reads data from `memory` and produces a `bytes memory` object, which the EVM then copies into `calldata` during the external call. Solidity does not permit manually constructing `calldata` structures; only `memory` and `storage` can be written to and manipulated by user code.

Therefore, even when the callee declares its parameter as `calldata`, the function type

still requires the caller to pass a `memory`-based value. The compiler then automatically handles the encoding and transfer of this data into `calldata`.

The process of preparing reference-type arguments for external calls can be summarized as follows:

- The ABI encoder expects the data to be in `memory`.
- If the data is already in `memory`, no action is required. Otherwise, a copy is made from `storage` or `calldata` to `memory`.
- ABI encoding produces a `bytes memory` payload.
- The EVM copies this payload from `memory` into `calldata` and attaches it to the message call.
- On the callee side:
  - If the parameter is declared as `calldata`, the data is ready to use directly.
  - If the parameter is declared as `memory`, Solidity generates a copy from `calldata` into `memory` so that the callee can work on a mutable copy.

This distinction between the caller's preparation and the callee's usage explains the `memory`/`calldata` mismatch in external function types.

Example: Difference Between Internal and External Memory Passing

The example below illustrates how passing data between `memory` references behaves differently in internal and external calls. When calling internally, `memory`-to-`memory` parameter passing preserves the reference (no copy), just like direct assignments between `memory` variables discussed earlier. In contrast, an external call creates an isolated copy of the data, preventing shared references across contract boundaries. Note that the same applies to `calldata`-to-`calldata` interactions.

```
contract C {
  function f(int[1] memory x) public pure {
    x[0] = 17;
  }

  function g() internal view returns (int, int) {
    int[1] memory a = [int(42)];
    this.f(a);            // external call: independent copy

    int[1] memory b = [int(42)];
    f(b);                 // internal call: pointer to the same memory

    return (a[0], b[0]);  // returns (42, 17)
  }
}
```

In this example:

- In the external call `this.f(a)`, the data is copied as described earlier. The callee receives a separate copy, so modifying `x` in `f` does not affect `a`.
- In the internal call `f(b)`, the same `memory` region is used directly—no copy is made—so the mutation to `x` in `f` affects `b`.
- The returned values demonstrate this distinction: `a[0]` remains unchanged, while `b[0]` reflects the mutation, confirming that internal calls preserve `memory` references

whereas external calls isolate them.

4.3.3 Scoping and Declarations

Solidity follows scoping rules similar to those of C99. A variable is visible from the point of its declaration until the end of the block in which it is declared, including any nested blocks, unless shadowed by a new declaration with the same name.

Declarations outside of any block, such as top-level or contract body declarations, are visible from anywhere in the contract—even before their point of declaration.

The following example illustrates these rules:

```
contract C {
  function f() public view returns (int) {
    int y = x;  // state variable x is visible before its declaration
    {
    // z = 0;  // error: z is not visible before its declaration
      int z = y;  // z = 1
      {
        int z = 0;  // shadows the outer z
      }
      return z;  // returns 1
    }
    // z = 1;  // error: z is not visible outside its block
  }

  int x = 1;
}
```

4.4 Contracts

4.4.1 Function Visibility

Visibility determines where functions can be accessed and is closely tied to how they can be called: internally, externally, or both. Solidity provides four visibility levels for functions:

- `private`: Accessible only internally within the contract where it is defined.
- `internal`: Accessible internally within the contract and any contract that inherits from it.
- `public`: Accessible both internally (contract and derived contracts) and externally through any contract.
- `external`: Accessible only externally within any contract.

Table 4.2 summarizes the accessibility of each function visibility level, broken down by access type and contract relationship.

Visibility	Internal Access			External Access
	Same Contract	Derived Contracts	Other Contracts	Any Contract
`private`	yes	no	no	no
`internal`	yes	yes	no	no
`public`	yes	yes	no	yes
`external`	no	no	no	yes

Table 4.2: Access rules for function visibility in Solidity

4.4.2 State Variable Visibility

Visibility controls where and how state variables can be accessed in Solidity. There are three visibility levels for state variables:

- `private`: Accessible only internally within the contract where it is defined.
- `internal`: Accessible internally within the contract and any contract that inherits from it.
- `public`: Accessible both internally (contract and derived contracts) and externally through any contract via an automatically generated getter function.

Note: Internal access to state variables does not involve a function call but rather direct access to the variable. Table 4.3 summarizes the accessibility of each state variable visibility level, broken down by access type and contract relationship.

Visibility	Internal Access			External Access
	Same Contract	Derived Contracts	Other Contracts	Any Contract
`private`	yes	no	no	no
`internal`	yes	yes	no	no
`public`	yes	yes	no	yes

Table 4.3: Access rules for state variable visibility in Solidity

4.4.3 `public` State Variable Getters

The compiler automatically generates getter functions for all `public` state variables. When the variable is accessed directly, it behaves like a state variable. However, when accessed via a contract instance (meaning as a member of an expression of contract type), as shown in the example below, it evaluates to an external function type marked `view`, as it does not modify contract state.

```
contract C {
  int public data = 42;
}

contract Caller {
  C c = new C();

  function f() public view returns (int) {
    return c.data();
  }
}
```

`public` state variables are part of the contract's interface and therefore cannot have internal function types, which are not representable externally. However, reference types are allowed because the compiler-generated getter functions avoid exposing contract `storage`: they return individual elements rather than the entire structure.

To achieve this, the following process is followed to determine the parameter and return types of the getter:

1. Begin with the declared type of the state variable.
2. While the type is a mapping or an ordinary array:
   - If it is a mapping, its key type is added as a parameter to the getter. This allows the function to access the value associated with a specific key. The process then continues with the mapping's value type.
   - If it is an array, a `uint256` parameter is added to represent the array index. This allows the function to access a specific element at that index. The process then continues with the array's element type.
3. Once the final, non-nested type is reached:
   - If it is a reference type, it is copied to `memory` and returned.
   - If it is a value type, it is returned directly.

Table 4.4 shows examples of getter function signatures for various state variable types. Note that state variables always live in `storage`.

State Variable Type	Getter Type
`int`	`function () external view returns (int)`
`mapping(address => bytes32)`	`function (address) external view returns (bytes32)`
`string[]`	`function (uint256) external view returns (string memory)`
`mapping(bool => address[])`	`function (bool, uint256) external view returns (address)`
`mapping(int8 => bytes[])[]`	`function (uint256, int8, uint256) external view returns (bytes memory)`

Table 4.4: Examples of getter function signatures for various state variable types

This mechanism exists to avoid expensive copies when returning an entire array and because mappings cannot be copied. If the entire array needs to be returned, it must be explicitly copied from `storage` to `memory`, as shown in the example below.

```
contract C {
  int[] public v = [int(0), 1, 2];

  // returns the entire array by copying from storage to memory
  function f() public view returns (int[] memory) {
    return v;
  }
}
```

4.4.4 State Mutability

State mutability in Solidity defines how a function interacts with the contract's persistent state. It indicates whether a function can read from or write to the blockchain and whether it can receive Ether. Each function is associated with a state mutability, and the restrictions are strictly enforced at compile time. Solidity defines four types of state mutability:

- `pure`: Functions marked `pure` cannot read from or modify the contract's state. This means they cannot access state variables or call functions that are not also marked `pure`. These functions are typically used for performing computations that depend only on their input parameters.
- `view`: Functions marked `view` are allowed to read from the contract's state but cannot modify it. They can access state variables and call other `view` or `pure` functions, but they cannot alter the blockchain. These functions are commonly used to return information about the contract's current state.
- `payable`: A `payable` function can read and modify contract state and is additionally permitted to receive Ether. Only functions marked `payable` can be the target of transactions that send Ether. This mutability is essential for implementing features like token purchases or deposits.
- *non-payable* (implicit): Functions that are not explicitly marked as `pure`, `view`, or `payable` are considered *non-payable* by default. These functions can read and modify state but cannot receive Ether. Any attempt to send Ether to such a function will result in the transaction being reverted.

Each state mutability keyword restricts the function's behavior in increasing degrees of freedom, and the compiler ensures that these rules are followed. This allows developers to make assumptions about the side effects of a function.

4.4.5 Inheritance and Linearization

Solidity supports multiple inheritance, allowing a contract to inherit from more than one base contract. Inheritance is specified using the `is` keyword following the contract's name:

```
contract A { }
contract B is A { }
contract C { }
contract D is C, B { }
```

When multiple contracts are inherited, Solidity must resolve the order in which base contracts are initialized and their members are accessed. This is particularly important when overridden functions and constructors are involved.

To manage this, Solidity uses C3 linearization, similarly to Python. C3 linearization defines a consistent ordering of base contracts in the inheritance hierarchy, ensuring that each contract appears only once in the linearization and that derived contracts precede their base contracts. This process guarantees a deterministic method resolution order (MRO) and avoids ambiguity caused by the Diamond Problem—a classic issue in multiple inheritance.

The result of this linearization is a list of contracts, from the most derived to the most base-like, including the contract itself. For the example above, the linearization of contract `D` is:

`L[D] = [D, B, A, C]`

Note that, unlike Python, Solidity requires the direct base contracts to be listed from most base-like to most derived.

Base Constructor Arguments

When a contract inherits from a base contract whose constructor requires arguments, those arguments must be provided explicitly by the derived contract. There are two ways to do this:

- Directly in the `is` clause, using argument syntax.
- Inside the derived contract's constructor, using modifier-style constructor invocation.

All required constructor arguments for all base contracts must be provided exactly once. Failing to provide them, or attempting to provide them multiple times—possibly through different inheritance paths—results in a compilation error.

```
contract A {
  constructor(int) { }
}

contract B is A(17) { }  // base constructor call

contract C is A {
  constructor() A(42) { }  // modifier-style base constructor call
}

// error: arguments to A provided twice via B and C
contract D is B, C { }
```

In this example, contract `D` inherits from both `B` and `C`, each of which attempts to initialize the constructor of `A`. The compiler reports an error because `A`'s constructor arguments are being provided along two separate paths, which would lead to ambiguity.

Chapter 5

SolGen

5.1 Overview

SolGen is a generator of semantically correct Solidity programs. It was developed with an emphasis on supporting a range of versions of the Solidity compiler. The oldest supported version is Solidity 0.5.0 and the newest one at the time of this writing is Solidity 0.8.30, although many of the more recent language features are currently not implemented. The project is available at https://github.com/alex2449/SolGen.

SolGen focuses solely on the static semantics of Solidity programs—that is, correctness as determined at compile time. It ensures that generated programs are well-formed with respect to typing, scoping, and other compile-time rules of the language. However, it does not attempt to guarantee that programs are free of undefined behavior and independent of unspecified behavior at runtime. This is a complex task that was not considered for the purposes of this thesis. This makes the implementation simpler but also limits its bug-finding capabilities, at least in the parts of the compiler that generate and optimize code. Therefore, differential testing is not possible with SolGen. Instead, the test oracle is solved by checking the compiler's response. We expect solc to successfully compile a given random program. If the compiler crashes or reports an error, assuming that the generator itself has no bugs, then there exists a bug in the compiler, which we can manually track down.

The general idea can be described as the reverse process of what a typical compiler front-end does. Instead of parsing source code into an AST, we construct an AST which is then "pretty-printed" back to Solidity code, by recursively converting each node to its equivalent syntactic form, following the language's grammar. Throughout generation, all decisions that require randomness are made with a biased coin toss or a weighted selection from a list of candidate elements. The rest of this Chapter describes the generation process. Note that the process described in this text considers only a subset of the language, and some constructs may be omitted for simplicity.

5.2 Generation Approach for Types, Expressions, and Statements

The bulk of random generation is concerned with three basic building blocks of most programming languages:

- Types, such as `int256`, `mapping(bytes17 => string[42])`, or `address[][10]`.
- Expressions, such as call or binary expressions. An expression evaluates to a value of a certain type.
- Statements, such as *block*, *if-else*, or *return* statements.

To generate these three building blocks, we implement three main functions:

- `generate_type()`: Generates a random type. This function is utilized in various situations.
- `generate_expression(type)`: Generates a random expression of a given type. This function is utilized for every syntax rule, or equivalently for every node in the AST, that contains an expression.
- `generate_statement()`: Generates a random statement. Similarly, this function is utilized for every node in the AST that contains a statement.

Each of these functions delegates to specialized functions that generate specific subcategories. For example, `generate_block_stmt()` generates a block statement, while `generate_binary_expr(type)` generates binary expressions. These specialized functions are used internally by the main generation functions in the following way:

1. **Candidate initialization**: A list of all productions (e.g., types, expressions, or statements) is assembled as initial candidates.
2. **Attempted generation**: A candidate is selected at random, and its corresponding generation method is invoked.
3. **Validation**: If the chosen production is not syntactically or semantically valid in the current context, an error is returned.
4. **Retry**: The invalid candidate is removed from the list, and another is selected.
5. **Success**: Once a valid production is generated, it is returned as the result.

5.3 Generating Types

5.3.1 Internal Representation of Types

Types are semantic constructs and not part of the formal grammar. The grammar defines type names—syntactic elements like `int256` or `address[][]`—along with data location specifiers (`memory`, `calldata`, `storage`). While these are represented as separate constructs in the syntax, they are interpreted together to form a complete type at the semantic level. For example, the syntax `bool[][] memory` is interpreted as the type:

`bool[] memory[] memory`

This should be read right-to-left, as: a dynamic array in `memory`, of dynamic arrays in

`memory`, of booleans. The data location at each level of the reference type is explicit. Internally, this would be represented as:

```
ArrayType(DataLocation::Memory, ArrayType(DataLocation::Memory, BoolType()))
```

Bound `storage` References vs Rebindable `storage` Pointers

Reference types with `storage` data location are internally distinguished between **bound references** and **rebindable pointers**:

- **Bound references** refer to a fixed, specific `storage` location (such as a state variable); they cannot be redirected to point elsewhere, and assignments to them modify the underlying storage.
- **Rebindable pointers** (e.g., local variables or function parameters), by contrast, can be reassigned to point to different `storage` locations. Assigning to the pointer itself does not change the data—only where the pointer refers.

Both are implemented as pointer values, but they differ in how the compiler treats them. Bound references are automatically dereferenced in most contexts, giving direct access to the underlying data. In contrast, rebindable pointers are not automatically dereferenced, so their pointer nature is preserved in expressions. This distinction explains why local `storage` variables (rebindable pointers) can be referenced in `pure` functions.

For `memory` and `calldata`, this distinction is unnecessary, as these locations always behave as rebindable pointers.

```
contract C {
  uint[][] x = [[42]];

  function f() public {
    uint[][] storage p = x;

    p[0] = [17];

    assert(x[0][0] == 17);
  }
}
```

In this example:

- The type of `x` is `uint[] storage reference[] storage reference`. This indicates a fixed `storage` location holding a dynamic array of dynamic arrays of `uint`.
- The type of `p` is `uint[] storage reference[] storage pointer`. This means `p` is a rebindable pointer to a dynamic array in `storage`, whose elements are themselves bound references to `uint[]` in `storage`.
- The assignment `p = x;` does not copy from `storage`—it simply makes `p` point to the same outer array as `x`. Reassigning `p` would not affect `x`.
- The expression `p[0]` has type `uint[] storage reference`. Assigning to it (as in `p[0] = [17];`) modifies the actual contents of storage—specifically, the inner array referenced by `x[0]`.

Mappings Classification

Although mappings behave like reference types at the language level, they are not classified as reference types internally. This is because mappings can only exist in `storage` and cannot be copied. As a result, they are treated separately from reference types during generation and are not explicitly annotated with a data location. In the remainder of this thesis, the term *reference type* excludes mappings. When referring to mappings, they will be mentioned separately.

5.3.2 Generating Mappings

Mapping type generation involves generating two types: the key type and the value type. We assume that a type is generated to be used in one of the following contexts:

- Array element type for an array with a known data location.
- Mapping key or value.
- Function parameter or return type for a function with a known calling convention.

Constraints.

- Mappings can only live in `storage`, therefore:
  - They can be parameters and return types only for internal function types, as external function types cannot receive or return `storage` references, as explained in Section 4.3.2.
  - They can be the element type only for arrays that live in `storage`.
- Mappings are not allowed as mapping keys.

Generation Procedure.

- If the context is a mapping key, generation fails.
- If the context is an array element for an array that does not live in `storage`, generation fails.
- If the context is a function parameter or return type for an external function type, generation fails.
- Otherwise, the key and value types are generated recursively by calling `generate_type()`. The new contexts are a mapping key and mapping value, respectively.

5.3.3 Generating Arrays

Array type generation involves the following:

1. Select array kind from:
   - Ordinary arrays (`T[N]` or `T[]`)
   - Dynamic byte arrays (`bytes`)
   - Strings (`string`)
2. Select data location from: `memory`, `storage`, and `calldata`. If `storage`, select between bound reference and rebindable pointer.

3. Generate the element type (for ordinary arrays).
4. Select whether static or dynamic (for ordinary arrays). If static, determine the array's length.

We assume that a type is generated to be used in one of the same contexts as for mappings.

Constraints.

- If a reference type is used as an array element, it must live in the same data location as the array. This is because composite types must be located fully in a single memory region; combinations such as `int[] memory[] storage pointer` are meaningless. If it is `storage`, it must be a bound reference.
- Ordinary arrays are not supported as mapping keys.
- If a reference type is used as a mapping key, it must live in `memory`. This is because mapping keys are not stored but hashed, and for reference types (e.g., `bytes`, `string`), the value to be hashed is expected to be in `memory`.
- If a reference type is used as a mapping value, it must live in `storage` and be a bound reference.
- If a reference type is used as a function parameter or return type, it must live in `memory`. This reflects a convention in Solidity's type system, as explained in Section 4.3.2.

Generation Procedure.

1. Select the array kind, by selecting between:
   - `bytes` and `string`, which are always allowed.
   - Ordinary arrays, if the context is not a mapping key.
2. Select the data location:
   - If the context is an array element, inherit the array's data location. If it is `storage`, select bound reference.
   - If the context is a mapping key, select `memory`.
   - If the context is a mapping value, select `storage` and bound reference.
   - If the context is a function parameter or return type:
     - If internal, select among `memory`, `calldata`, `storage`. If `storage` was selected, select rebindable pointer.
     - If external, select `memory`.
3. Generate the element type (for ordinary arrays): recursively call `generate_type()`, adjusting context. The new context will be an array element for the selected data location.
4. Select static or dynamic (for ordinary arrays) without restrictions: if static, randomly choose a positive length.

5.3.4 Generation Example

Suppose we want to generate a type `T` to be used as a parameter in the following function type:

```
function (bytes32, T) external view returns (bool)
                   ↳ What can be placed here?
```

In this case, the context for generating `T` is a function parameter for an external function type. The following steps may be followed to generate `T`:

1. Suppose the generator attempts to generate a mapping for `T`: this fails because it would require an external function that receives a mapping.
2. Suppose the generator attempts to generate an array for `T`:
    1. Select its kind: all three kinds are allowed in this context. Suppose an ordinary array is selected.
    2. Select its data location: since the context is a function parameter for an external function type, select `memory`.
    3. Select either a dynamic array or a static array of any length: suppose dynamic array.
    4. Generate the element type of `T`, `E`, recursively: the context for generating `E` will be an array element for an array in `memory`.

        ```
        function (bytes32, E[] memory) external view returns (bool)
                           ↳ What can be placed here?
        ```

        1. Suppose the generator attempts to generate a mapping: this fails because it would require a mapping (that can live only in `storage`) as an array element for an array in `memory`.
        2. Suppose the generator attempts to generate an array for `E`:
            1. Select its kind: similarly, suppose `bytes` is selected.
            2. Select its data location: since the context is an array element for an array in `memory`, select `memory`.

The resulting type `T` is: `bytes memory[] memory`.

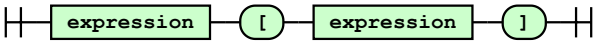
5.4 Generating Expressions

A big part of the generation process involves generating a well-typed program that is accepted by the type checker. The generation of expressions is mostly directed by the expected result type and context-specific constraints, such as whether the expression is used as an l-value.

5.4.1 Generating Indexed Accesses

The syntax of an index access expression is given by:

rule index-access-expr



The generation of this expression involves generating two subexpressions: the base and the index, which requires the construction of two types, one for each subexpression. We consider the following cases for the type of the base for which the expression is valid:

- Mappings
- Arrays (ordinary and `bytes`)
- Fixed-size byte arrays (`bytesN`)

Cases are selected at random until one succeeds or all are exhausted. Each case is described separately below.

Mappings

Constraints.

- The type of the expression is the mapping's value type.
- The expression is not an l-value if assigning to it would require copying a mapping, which is disallowed in Solidity (see below).
- The expression cannot be used if the context is a `pure` function. This is because mappings always live in `storage`.
- The expression cannot be used as an l-value if the context is a `pure` or `view` function, for the same reason.
- The index's type must be implicitly convertible to the mapping's key type.

When Assignment Requires Copying a Mapping

Expressions that, when assigned to, require the copying of a mapping are not considered l-values. This occurs in the following cases:

- The expression is of mapping type and it is not a local variable or parameter.
- The expression is a bound `storage` reference that contains a mapping in one of the following ways:
  - It is an array whose element type is or recursively contains a mapping.
  - It is a struct with a member whose type is or recursively contains a mapping.

```
contract C {
  struct S {
    mapping(int => int)[] m;
  }

  mapping(int => S) v1;
  S v2;

  function f() internal returns (S storage rp) {
    // error
    v1[0] = v2;

    rp = v2;
  }
}
```

In this example:

- The type of `v1` is `mapping(int => S storage reference)`.
- The assignment `v1[0] = v2;` is invalid because the type of `v1[0]` is `S storage reference`, and therefore this assignment requires copying `v2`, which also has type `S storage reference`.
  - To copy a value of type `S`, Solidity must copy its member `m`, which is an array.
  - Copying the array `m` would require copying its elements.
  - But the elements are mappings, which cannot be copied.
- The assignment `rp = v2;` is valid because the type of `rp` is `S storage pointer`; it only causes `rp` to point to `v2`'s `storage` slot.

Generation Procedure.

- If the requested type is not a valid mapping value type, generation fails. Mapping value types include:
  - Value types
  - Reference types that are bound `storage` references
  - Mappings
- If an l-value is requested:
  - If the requested type is a mapping, generation fails. This would require an l-value of mapping type of the form $e_1[e_2]$, which is not possible.
  - If the requested type is a bound `storage` reference that contains a mapping, this should not occur and indicates a bug in the generation logic.
- If the context is a `pure` function, generation fails.
- If the context is a `view` function and an l-value is requested, generation fails.
- Construct the base's type:
  - The mapping's key type is generated randomly using the `generate_type` function.
  - The mapping's value type is set to the requested type.
- Construct the index's type: it is generated randomly as a type that is implicitly convertible to the mapping's key type (see Section 5.4.3).

Arrays

Constraints.

- The type of the expression is:
  - The array's element type for ordinary arrays.
  - `bytes1` for dynamic byte arrays (`bytes`).
- The expression is not an l-value if:
  - Assigning to it would require copying a mapping.
  - The array's data location is `calldata`, which is read-only.
- The expression cannot be used if the array's data location is `storage` and the context is a `pure` function.
- The expression cannot be used as an l-value if the array's data location is `storage` and the context is a `pure` or `view` function.
- The index's type must be implicitly convertible to `uint256`.

Generation Procedure.

- If the requested type is not a valid array element type or `bytes1`, generation fails. Array element types include:
  - Value types
  - Reference types that are not rebindable `storage` pointers
  - Mappings
- If an l-value is requested:
  - If the requested type is a mapping, generation fails (would require an l-value of mapping type of the form $e_1[e_2]$).
  - It should not be possible for the requested type to be a bound `storage` reference that contains a mapping.
- Construct the base's type:
  - Select the array kind, by selecting between:
    - An ordinary array, which is always allowed.
    - `bytes`, which is allowed if the requested type is `bytes1`.
  - Select the data location:
    - The set of candidate data locations is initially empty.
    - If an ordinary array has been selected and the requested type is a reference type or a mapping, then we add the requested type's data location or `storage` respectively to the candidate set. This is because composite types must be located fully in a single memory region.
    - Otherwise, we add all three data locations (`memory`, `storage`, `calldata`).
    - Apply the following restrictions:
      - Remove `calldata` if an l-value is requested.
      - Remove `storage` if the context is a `pure` function, or a `view` function and an l-value is requested.
    - If the set of data locations is empty, generation fails.

    - Otherwise, randomly select a data location. If the selected data location is `storage`, select either a bound reference or a rebindable pointer.
  - If an ordinary array has been selected, set the element type to the requested type.
  - If an ordinary array has been selected, randomly choose between static or dynamic. If static, randomly select a length to be a positive integer.
- Construct the index's type: it is generated randomly as a type that is implicitly convertible to `uint256`.

Fixed-size Byte Arrays (`bytesN`)

Constraints.

- The type of the expression is `bytes1`.
- The expression is not an l-value.
- The index's type must be implicitly convertible to `uint256`.

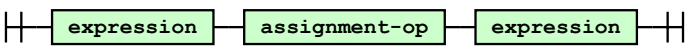
Generation Procedure.

- If the requested type is not `bytes1`, generation fails.
- If an l-value is requested, generation fails.
- Construct the base's type: select a valid fixed-size bytes type (`bytes1`, `bytes2`, …, `bytes32`).
- Construct the index's type: it is generated randomly as a type that is implicitly convertible to `uint256`.

5.4.2 Generating Assignments

In Solidity, assignments are expressions that cause side effects and evaluate to a value of a certain type. There are three kinds of assignments—ordinary, compound, and tuple—but we will consider only ordinary assignments. The syntax of an assignment expression is given by:

rule assignment-expr



Constraints. In a valid assignment:

- The type of the expression is the type of the first operand.
- The expression is not an l-value.
- The first operand must be an l-value.
- The type of the second operand must be implicitly convertible to the type of the first operand.
- Assignments that modify contract state are disallowed in `pure` or `view` functions.

Generation Procedure.

- If an l-value is requested, generation fails.
- If the requested type is a bound `storage` reference:
  - If it contains a mapping, generation fails. Since the requested type will also be the type of the first operand, this would require an l-value of a bound `storage` reference type that contains a mapping, which is not possible.
  - If the context is a `view` function, generation fails. This assignment would modify `storage`, which is not permitted by `view` functions. No check is required for `pure` functions, as in this case, such an expression should not have been requested in the first place.
- Set the type of the first operand to the requested type and request an l-value for it.
- Construct the second operand's type: it is generated randomly as a type that is implicitly convertible to the first operand's type.

5.4.3 Implicit Type Conversions

The compiler checks whether a type can be implicitly converted to a destination type in various contexts:

- Assignment context: in an assignment `V = E`, when implicitly converting from the type of `E` to the type of `V`.
- Argument conversion context: when implicitly converting the type of an argument to the corresponding parameter type, in all calling contexts (function calls, modifier invocations, inheritance specifiers, etc.).
- Return context: when converting the type of the expression in a return statement to the function's return type.
- Other contexts: used for a small number of additional conversions.

In the context of generation, we need to generate an expression whose type is implicitly convertible to a given destination type. This is the reverse of what the compiler does: instead of checking whether a conversion is valid, we start from the destination type and construct a suitable source type.

Function Types

Conversion Semantics. A function type is implicitly convertible to another destination function type if:

- The destination has the same calling convention, parameter types, and return types as the source.
- The following state mutability conversions are allowed, as illustrated in Figure 5.1:
  - Only `pure` functions can be used where a `pure` function type is expected.
  - `pure` and `view` functions can be used where a `view` function type is expected.
  - `pure`, `view`, *non-payable*, and `payable` functions can be used where a *non-payable* function type is expected. The reason `payable` functions are

allowed here is because the compiler only prevents Ether from being sent through a *non-payable* function pointer—it does not enforce that the referenced function itself is *non-payable*.

- Only `payable` functions can be used where a `payable` function type is expected. This restriction exists because `payable` function pointers can be used to send Ether. If such a pointer referenced a `pure`, `view`, or *non-payable* function, it could cause a runtime error when Ether is sent.

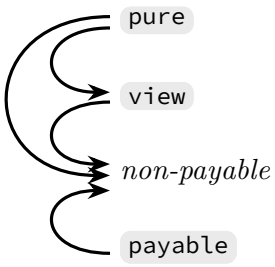


Figure 5.1: Allowed implicit conversions between function state mutabilities

Generation. The following steps are followed to generate a source function type given a destination function type:

- Set the source to a function type with identical calling convention, parameter types, and return types to the destination.
- Choose a compatible state mutability based on the destination:
  - If the destination is `pure`, select `pure`.
  - If the destination is `view`, select `pure` or `view`.
  - If the destination is *non-payable*, select any of: `pure`, `view`, *non-payable*, or `payable`. However, `payable` is only valid here if the calling convention is external, as only external functions can be `payable`. This is not a restriction of the conversion semantics, but a constraint imposed by the Solidity type system—internal function types cannot be `payable` because internal calls do not support Ether transfer.
  - If the destination is `payable`, select `payable`.

Array Types

Conversion Semantics. This section describes the semantics—what is allowed and what happens during implicit conversions to array types, depending on the destination's data location. It is this implicit conversion that determines the assignment semantics for reference types described in Section 4.3.1.

Note that the behavior for `memory` and `calldata` applies only to assignments, internal function calls, and similar contexts—not to external calls, which always create an isolated copy of the data.

1. If the destination is a bound `storage` reference:

- The source may have any data location; if it is `storage`, both bound references and rebindable pointers are allowed.
- In all cases, an independent deep copy of the data is created.
- The source's element type must be implicitly convertible to the destination's element type (not necessarily identical).
- If the destination is a dynamic array, the source must be a dynamic array or a static array of any length.
- If the destination is a static array, the source must be a static array of equal or smaller length.

2. If the destination is a rebindable `storage` pointer:
   - The source must be in `storage` (bound reference or rebindable pointer).
   - Other data locations are not allowed, as this would require the creation of a temporary in `storage` just to give the `storage` pointer something valid to point to, which is not supported.
   - No data is copied; only a pointer is assigned.
   - Types must be identical, except for the bound reference/rebindable pointer distinction.
3. If the destination is in `memory`:
   - The source may have any data location; if it is `storage`, both bound references and rebindable pointers are allowed.
   - If the source is in `storage` or `calldata`, an independent deep copy of the data is created.
   - If the source is in `memory`, only a pointer is copied.
   - Types must be identical, excluding the data location.
4. If the destination is in `calldata`:
   - The source must be in `calldata`.
   - Only a pointer is copied.
   - Types must be identical.

Generation. This section describes how to generate a source array type, given a destination array type, such that the source is implicitly convertible to the destination.

1. If the destination is a bound `storage` reference:
   - Select any data location: `storage`, `memory`, or `calldata`. If the selected location is `storage`, choose between bound reference or rebindable pointer.
   - If the destination is a dynamic array, select a dynamic array or a static array of any length.
   - If the destination is a static array, select a static array of equal or smaller length.
   - Recursively generate the source's element type to be implicitly convertible to the destination's element type.
2. If the destination is a rebindable `storage` pointer:
   - Set the data location to `storage` and select either a bound reference or a

rebindable pointer.
- Set the source type to be identical to the destination type, except for the bound reference/rebindable pointer distinction.

3. If the destination is in `memory`:
   - Select any data location: `storage`, `memory`, or `calldata`. However, `storage` should only be considered if the context allows reading from storage (i.e., not within a `pure` function), as this conversion copies the array's contents from `storage` to `memory`.
   - If the selected location is `storage`, choose between a bound reference and a rebindable pointer.
   - Notably, the Solidity compiler exhibits a bug, as it currently permits conversions from rebindable `storage` pointers to `memory` even in `pure` functions—despite this requiring a read from `storage`. This occurs because the compiler analyzes each expression independently and does not evaluate whether the conversion semantics involve side effects such as storage reads.

```
        function f(int[] storage p) internal pure {
          int[] memory m;
          m = p;  // allowed!
        }
```

     For the same reason, conversions from bound `storage` references to rebindable `storage` pointers could be allowed even when reading from `storage` is disallowed. Since such conversions only copy a pointer and do not access `storage`, they are side-effect free. However, the compiler currently rejects them.
   - Set the source type to be identical to the destination type (in the selected data location).
4. If the destination is in `calldata`, set the source type to be identical to the destination type.

Generation Example

Given the destination type:

`function () internal view returns (int)[] storage reference`

we wish to construct a source type that is implicitly convertible to it. The following steps are followed:

1. Determine data location: the destination is a bound `storage` reference, so any data location is allowed for the source. Suppose we select: `memory`.
2. Select array kind: the destination is a dynamic array, so the source may be either a dynamic array or a static array of any length. Suppose we select: static array of length 10.
3. Construct the element type: this step is recursive. Given the destination type: `function () internal view returns (int)`

we construct a source type that is implicitly convertible to it:

1. Keep the calling convention, parameter types, and return types identical to the destination.
2. Select a state mutability that is allowed by the conversion rules. Since the destination is `view`, valid options are `pure` and `view`. Suppose we select: `pure`.
   The resulting function type is:
   `function () internal pure returns (int)`

Therefore, the constructed source type is:

`function () internal pure returns (int)[10] memory`

This process can be seen as an exploration of structural mutations to a type, demonstrating how the generator systematically explores valid execution paths through the type conversion rules. Over multiple runs, the generator probabilistically covers the full space of conversion-permissible source types for a given destination.

5.5 Generating Statements

Statements define control flow, side effects, and structure within a Solidity program. Statement generation is relatively straightforward, and a few semantic constraints must be enforced to ensure validity. Statements are grouped into the following categories:

- **Control flow statements:** These include `if`, `while`, *do-while*, and `for` statements. For these, the type of the condition must be `bool`.
- `continue` **and** `break` **statements:** These are only valid within a loop (`while`, *do-while*, or `for`).
- **Return statements:** terminate execution of a function and optionally return a value. The expression, if present, must be compatible with the function's return types. For functions with no return types, a bare `return;` is also allowed.
- **Variable declaration statements:** introduce new local variables, optionally with initializers. If present, the initializer's type must be implicitly convertible to the declared type. The variable is visible from the point of its declaration until the end of the block where it is defined.
- **Block statements:** consist of a sequence of statements enclosed in `{ }` and introduce a new lexical scope for local variables. To support generation of usable and correctly scoped local variables, the following strategy is used:
  - The statements in a block are generated in reverse order. This counterintuitive approach ensures that when an expression or statement refers to a local variable, that variable will be declared later (i.e., higher) in the block without violating scoping rules.
  - A stack of scopes is maintained during generation. Each block is associated with one scope level.

  - When an expression requests a reference to a local variable, it may create a new variable and register it in a scope.
  - Later, when a variable declaration statement is chosen for generation, some variables from the current scope are declared and removed from visibility (i.e., moved from ”pending” to ”declared”).
  - At the end of the generation, any remaining undeclared variables are emitted as declarations.
- **Expression statements:** consist of a standalone expression, typically used for its side effects (e.g., assignments or function calls), but any valid expression of any type may be used, even if it has no side effects.
- **Placeholder statements (`_;`):** These are allowed only within the body of a modifier. They indicate where the modified function's body will be inserted when the modifier is applied.
- **Emit statements:** are used to trigger events. These are disallowed within `pure` or `view` functions, as event emission is considered a state-modifying operation.

5.6 Generating Declarations

It is likely that the generation of some nodes might not be immediately possible. For example, if an *identifier* expression—an expression that references a declaration—is chosen to generate an expression of some type, a suitable declaration may not exist or it may not be visible at the current generation point. There are two strategies at that point:

- **Backtracking:** The generator can abandon the attempt to generate the *identifier* expression, remove it from the list of candidate expressions, and retry with another candidate. If no suitable alternatives remain, generation fails for that expression node. This failure then propagates upward, requiring parent nodes to also be discarded. Any contextual changes made during generation must be reverted.

  Consider the example in Figure 5.2. The generator attempts to generate a call to function `f`, which accepts two parameters. The first argument (`new A()`) is successfully created, causing contract `A` to become a dependency of contract `B`, since contracts are dependent on contracts that they create via `new`. However, the second argument cannot be generated because there is no matching expression of the required function type in scope. Consequently, the entire function call is discarded, and contract `A` is no longer considered a dependency of contract `B`. This is important because the contract dependency graph must not be cyclic.
- **Directed Generation:** Alternatively, the generator can step out of the current generation context and generate the required dependencies first. This is known as *directed generation* and it is the one implemented.

```
contract A {}

contract B {
  function f(A p1, function () internal pure p2) internal {
    // ...
  }

  function g() internal {
    f(new A(), <ERROR>);
  }
}
```

Figure 5.2: Example of generation failure and backtracking

While contract-level declarations are also generated in a top-down manner according to the language grammar, it is also necessary to generate them as a dependency for the generation of some expression. This ensures that `generate_expression` always succeeds. Otherwise, producing valid expressions for certain complex types would be extremely unlikely—even in large test cases. For instance, consider a function type with two parameters and two return parameters. If each type is restricted only to Solidity's 64 integer types, the total number of unique function types is ∼17 million. Randomly producing an expression of such a type from the existing context is practically impossible.

This section covers the generation of contract-level declarations as a dependency for generating some expression. There are two kinds of declarations that are generated in this manner: function definitions and state variable declarations.

5.6.1 Generating Function Definitions

Referencing a function directly by name (*rule* identifier-expr) results in an expression of an internal function type. Therefore, a function definition may be generated as a dependency of *identifier-expr* when:

- The requested type is an internal function type.
- An l-value is not being requested, as function identifiers are not assignable.

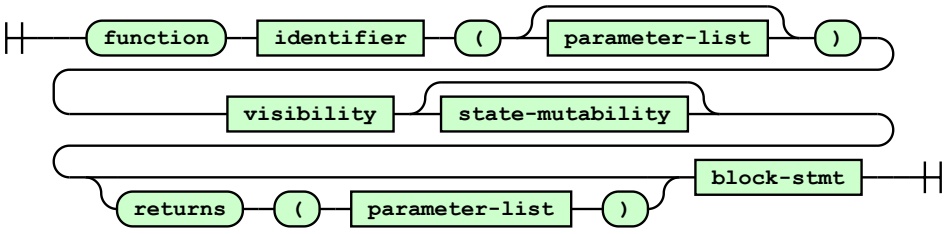


Figure 5.3: Grammar rule for function definitions

The process of generating a function definition, shown in Figure 5.3, involves the following steps:

1. Generate a unique identifier in order to avoid name conflicts.

2. Select visibility (one of `private`, `internal`, `public`, `external`):
   - `private` and `internal` are always allowed in this context.
   - `public` is allowed if every parameter type and return type of the requested function type can be used externally. A type cannot be used externally if:
     - It is or contains an internal function type.
     - It is a reference type that lives in `storage`.
     - It is a mapping.
   - `external` is disallowed in this context because external functions can only be called externally and are not visible by name.
3. Select state mutability:
   - If the requested function type is `pure`, `view`, or `payable`, add the corresponding keyword.
   - If the requested function type is *non-payable*, omit the state mutability (defaults to *non-payable*).
4. Select enclosing contract: the contract in which the function is placed is chosen as follows:
   - If the selected visibility is `private`, select the current contract.
   - Otherwise, select any contract in the current contract's linearization.
   - Insert the function at any position within the selected contract's body.
5. Construct the parameter and return parameter lists: each parameter type and return type of the requested function type is converted to a syntactic parameter declaration (see Figure 5.4). This involves the following steps:
   - Construct a type name—a syntactic representation of the type from its internal form. For example, the type `int[] memory[] memory` is converted to the type name `int[][]`. This step must follow the selection of the enclosing contract (which will be used as the scope), as the construction of a type name is a context-sensitive process (for user-defined types).
   - Select data location:
     - If the type is a value type, omit the data location.
     - If the type is a reference type, select its data location. Note that this is not always necessary; in the context of external function types (not relevant here), parameter and return types in `memory` may arbitrarily use `calldata` in the parameter declaration.
     - If the type is a mapping, select `storage`.
   - Generate a unique identifier for the parameter name to avoid name conflicts.
6. Generate a block statement to serve as the function's body.

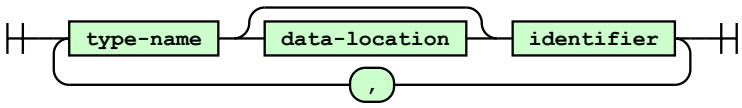


Figure 5.4: Grammar rule for parameter lists

5.6.2 Generating State Variable Declarations

Referencing a state variable as a member of an expression of a contract type results in an expression of an external function type—the state variable's getter. Therefore, a state variable declaration may be generated as a dependency of *member-access-expr* when:

- The requested type is a function type that is a valid getter type. A valid getter type must be a function type of the form:
  `function (T₁, T₂, …, Tₙ) external view returns (T)`
  where each `Tᵢ` is `uint256` or a valid mapping key type, and `T` is a value type or `bytes memory` or `string memory`. Mapping key types include:
  - Value types, except function types and `address payable`
  - `bytes memory` and `string memory`
- An l-value is not requested.

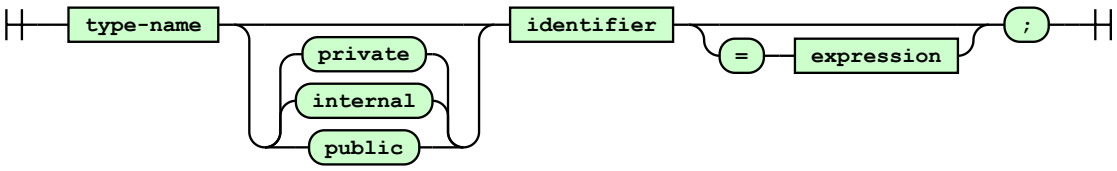


Figure 5.5: Grammar rule for state variable declarations

The process of generating a state variable declaration, shown in Figure 5.5, involves the following steps:

1. Generate a unique identifier in order to avoid name conflicts.
2. Select visibility: the visibility must be `public` for the state variable to be externally accessible and have a getter function.
3. Select enclosing contract:
   - Select any existing contract, as `public` state variables are externally accessible from any contract.
   - Insert the state variable at any position within the selected contract's body.
4. Construct the type name. To do that, the variable's type must first be determined using the getter's type:
   - Start with the return type `T` from the getter.
     - If it is `bytes memory` or `string memory`, set the current type to

`bytes storage reference` or `string storage reference`, respectively.
- Otherwise, set the current type to `T`.

- For each parameter `Tᵢ` from right to left:
  - If `Tᵢ` is not `uint256`, set the current type to `mapping(Tᵢ, current)`.
  - Otherwise, set the current type to one of:
    - `mapping(uint256, current)`
    - `current[] storage reference`
    - `current[N] storage reference`, where `N` is any positive integer.

The resulting type can be used as the state variable's type. For example, starting with:

`function (uint256, address) external view returns (string memory)`

we can construct the type:

`mapping(address => string storage reference)[] storage reference`.

The constructed type can now be converted to a type name—its equivalent syntactic form. For the previous type, this would be:

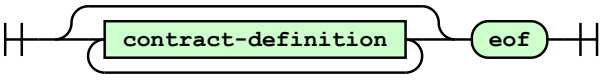
`mapping(address => string)[]`

5. Generate an optional initializer:
   - An initializer is not allowed if the variable's type is a mapping or if it is a bound `storage` reference that contains a mapping, as this would require copying a mapping.
   - Otherwise, an initializer may be generated. Its type must be implicitly convertible to the variable's type, and it is generated as an implicit conversion source with the destination being the variable's type.

5.7 Generating a Solidity Source File

At the highest level, a Solidity source file consists of a sequence of contract definitions.

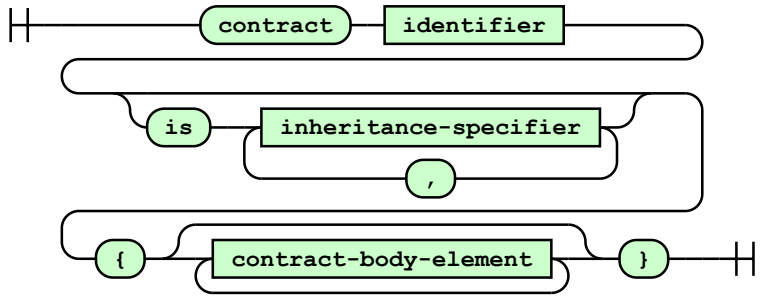
rule source-unit



Each contract definition includes:

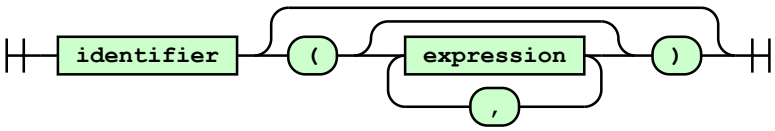
- A name
- An optional list of inheritance specifiers
- A body containing elements such as function definitions and state variable declarations

rule contract-definition



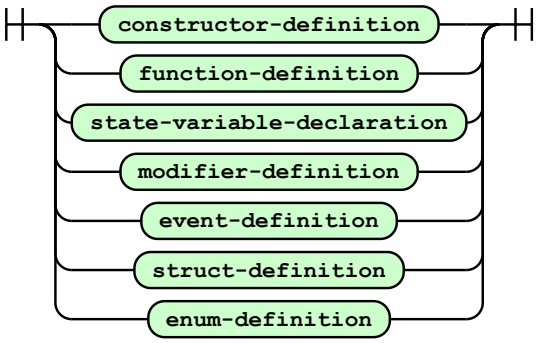
An inheritance specifier names a contract to inherit from and may optionally include a list of expressions to be passed as arguments to that contract's constructor.

rule inheritance-specifier



The body of a contract defines its functionality. It consists of a list of elements such as function definitions, state variable declarations, etc.

rule contract-body-element



Empty Contract Generation

We do not generate one complete contract after another. The reason is that Solidity allows symbols to be referenced before their declaration. For example, the signature of a function in the first contract may receive a type that is defined later, as shown in the example below:

```
contract C1 {
  function f1(C2 p1) internal view {
    // ...
  }
}

contract C2 {
  // ...
}
```

Therefore, as a first step, we create a list of empty contracts. For each contract, we only make a decision on whether the contract is abstract and whether it will have an explicit constructor definition. This needs to be known, as will be shown later. At this point, the program might look like this:

```
contract C1 {
}

contract C2 {
}
```

Inheritance and Contract Body Elements

Once this is complete, we iterate over the list of contracts and, for each contract:

- Determine the list of contracts that will appear in the contract's inheritance specifier list (the base contracts).
- Create a list of empty contract body elements for the same reason: we do not assign types to state variables, function parameters, etc. yet because these types might not exist at this point. For constructors, we select whether they receive parameters, as this will be needed later.

Base Contracts

To determine the list of contracts, if any, that will appear in a contract's inheritance specifier list, we randomly select candidate lists formed by randomly combining contracts from the set of previously declared contracts and use a candidate list if:

- The resulting inheritance graph can be linearized under the C3 algorithm.
- All required constructor arguments can be resolved without ambiguity.

The second restriction exists because certain inheritance scenarios must be disallowed when base constructor arguments cannot be resolved unambiguously. For example:

```
contract A {
  // assume <param-list>.size > 0
  constructor(<param-list>) { }
}
contract B is A[(<call-arguments)] { }
contract C is A[(<call-arguments)] { }
// error
contract D is B, C /*???*/ { }
```

```
contract A {
  constructor() public { }
}
contract B is A { }
contract C is A { }
// this is valid
contract D is B, C { }
```

In the first program, contract `A` defines a constructor that receives parameters. Contract `D` examines whether it can inherit from contracts `B` and `C`. This is not possible: because `A`'s constructor requires at least one parameter, contracts `B` and `C` must each provide these parameters when inheriting from `A`, assuming that contracts `B` and `C` are not abstract. Inheriting from both would cause `A`'s constructor arguments to be provided twice, creating ambiguity. In the second program, no such problem exists because `A`'s constructor receives no arguments.

To examine whether all required constructor arguments can be resolved without ambiguity, we need to determine which base contracts must be supplied with constructor arguments directly. For example:

```
contract A {
  constructor(int) { }
}

contract B is A(0) { }
contract C is A, B { }
```

Here, contract `A` has a constructor that takes one parameter. Contract `B` inherits from `A` and therefore provides the argument `0`. Contract `C`, which inherits from both `A` and `B`, must not also pass an argument to `A`'s constructor, since it is already provided via `B`; otherwise `A` would receive constructor arguments twice. Notably, this simple randomly generated program triggered an ICE in some older compiler versions.

In general, the contract body elements of previously declared contracts can affect which inheritance scenarios are valid. Another example (which currently does not affect the generator) is the following:

```
contract C1 {
  function f() public pure virtual { }
}

contract C2 {
  function f() public pure { }
}

contract C3 is C1, C2 /*???*/ {
}
```

Contracts `C1` and `C2` define a function with the same name `f` and parameter types. When `C3` attempts to inherit from `C1` and `C2`, it must override both `C1.f` and `C2.f`. This is because, when a contract inherits multiple functions with the same name and parameters, it must override them all. However, `C2.f` is not marked `virtual`, so it cannot be overridden. Assuming backpatching is not available to retroactively mark `C2.f` as `virtual` (which we avoid in general), `C3` cannot inherit from both `C1` and `C2`.

Additionally, inherited contracts also:

- Dictate which declarations **must** be present in a contract body. For example, when inheriting from an interface (interfaces in Solidity are similar to interfaces from other object-oriented programming languages), a contract must implement all of the interface's functions.
- Determine which symbols are in scope and therefore may not be redeclared. For example, two state variables with the same name cannot be in scope simultaneously.

For these reasons, inheritance and the symbols that will be present in a contract's body are determined contract by contract. We do not first produce a valid inheritance graph for all empty contracts and then populate their bodies.

Contract Body Elements

In order to create a list of empty contract body elements, a few constraints must be enforced, specifically relating to the selected attributes/specifiers of each symbol. For example:

- Only `public` or `external` functions can be `payable`.
- Constructors can only be *non-payable* or `payable`.
- Only one constructor can be defined per contract.

At this point, the program might look like this:

```
contract C1 {
  constructor(<param-list>) public payable <block-stmt>

  function f1(<param-list>) public payable [returns (<param-list>)] <block-stmt>

  <type-name> private v1;

  event E1(<param-list>);
}

contract C2 is C1[(<call-arguments>)] {
  function f2(<param-list>) internal view [returns (<param-list>)] <block-stmt>

  struct S1 {
    <struct-member-list>
  }
}
```

Completing Struct Definitions

Next, we complete existing struct definitions. The structure of structs must be known before completing other declaration headers. For example, once the struct `S1` of contract `C2` is populated with a mapping, its type will not be considered for parameters of, e.g., external functions (`S1` can now only live in `storage`).

During this step, we must ensure that no illegal cycles are created between structs. For example:

```
contract C {
  struct S1 {
    S2 m1;
  }

  struct S2 {
    S1[5] m1;
  }

  struct S3 {
    S1[] m1;
  }
}
```

Here, to determine the size of `S1`, the size of its member `m1` of type `S2` must be known. To determine the size of `S2`, the size of `S1[5]` must be known, which again requires the size of `S1`, creating an illegal cycle. By contrast, there is no cycle between `S1` and `S3` because the size of a dynamic array is independent of the size of its elements.

At this point, the program might look like this:

```
contract C1 {
  constructor(<param-list>) public payable <block-stmt>

  function f1(<param-list>) public payable [returns (<param-list>)] <block-stmt>

  <type-name> private v1;

  event E1(<param-list>);
}

contract C2 is C1[(<call-arguments>)] {
  function f2(<param-list>) internal view [returns (<param-list>)] <block-stmt>

  struct S1 {
    int m1;
    mapping(address => bytes[]) m2;
  }
}
```

Type Name Generation

In the following step, we walk the AST and generate type names for any construct that contains one. This includes functions, constructors, modifiers, and events, all of which contain type names in their headers, and state variable declarations. The `generate_type()` function is used here: it generates a type suitable for the given context. For example:

- `public` or `external` functions cannot receive or return reference types in `storage`, mappings, internal function types, or types that contain them.
- The type of a `public` state variable must not be, or expose (contain), internal function types.

The generated type is then converted to a type name and inserted into the AST. At this point, the program might look like this:

```
contract C1 {
  constructor(bytes memory p1) public payable <block-stmt>

  function f1(int p1) public payable <block-stmt>

  C2.S1 private v1;

  event E1(int[2] p1, bytes32 p2);
}

contract C2 is C1[(<call-arguments>)] {
  function f2() internal view returns (int[] storage rp1) <block-stmt>

  struct S1 {
    int m1;
    mapping(address => bytes[]) m2;
  }
}
```

Statement and Expression Generation

Finally, we walk the AST once again and generate statements or expressions for any construct that contains one. This includes function bodies, which involve the generation of a block statement, inheritance specifiers and modifier invocations, which may involve the generation of expressions, and state variables, which may involve the generation of an expression (the initializer). This step may cause the generation of additional function definitions and state variable declarations, as explained in Section 5.6. The program below shows the resulting program.

```
contract C1 {
  constructor(bytes memory p1) public payable { }

  function f1(int p1) public payable {
    emit E1([int(-1), p1], "");
  }

  C2.S1 private v1;

  event E1(int[2] p1, bytes32 p2);
}

contract C2 is C1("abc\n") {
  function f2() internal view returns (int[] storage rp1) {
    return v2;
  }

  int[] private v2;  // injected declaration

  struct S1 {
    int m1;
    mapping(address => bytes[]) m2;
  }
}
```

Chapter 6

Implementation

This chapter provides an overview of the internal implementation of the generator, focusing on the key data structures and representative generation procedures. The goal is to enable contributors to understand, modify, or extend the tool effectively.

6.1 Overview

The generator is implemented in C++ and mimics certain components of the Solidity compiler, particularly its Abstract Syntax Tree (AST) and type system. The core components of the generator are:

- **AST classes:** Model the Solidity source code structure.
- **Type system:** Defines how types are represented internally.
- `Context` **class:** Encapsulates contextual information during generation.

6.2 Abstract Syntax Tree

The AST represents the structure of the Solidity source code being generated. It is defined as a class hierarchy rooted at the `SourceUnit` class. The AST is composed of four main constructs, each represented by an abstract base class:

- `class Declaration`: Base class for all declarations (e.g., `ContractDefinition`, `FunctionDefinition`).
- `class TypeName`: Base class for all type names (e.g., `ArrayTypeName`, `FunctionTypeName`).
- `class Statement`: Base class for all statements (e.g., `IfStatement`, `ReturnStatement`).
- `class Expression`: Base class for all expressions (e.g., `IndexAccessExpression`, `CallExpression`).

Other AST nodes include `SourceUnit`, `InheritanceSpecifier`, and `ModifierInvocation`. Any node of the AST that is required to do so implements generation-related methods and a method to pretty-print itself.

Example

```
class SourceUnit {
  std::vector<ContractDefinition *> contracts;
};
```

The `SourceUnit` contains a list of contract definitions. A contract definition is represented as:

```
class ContractDefinition : public Declaration {
  std::string name;
  std::vector<InheritanceSpecifier *> inheritance_specs;
  std::vector<Declaration *> body;
};
```

where `InheritanceSpecifier` is defined as:

```
class InheritanceSpecifier {
  std::string name;  // the name of the contract to inherit from
  std::vector<Expression *> arguments;
};
```

6.3 Type System

The type system is modeled as a class hierarchy rooted at an abstract base class `Type`, from which all concrete types inherit:

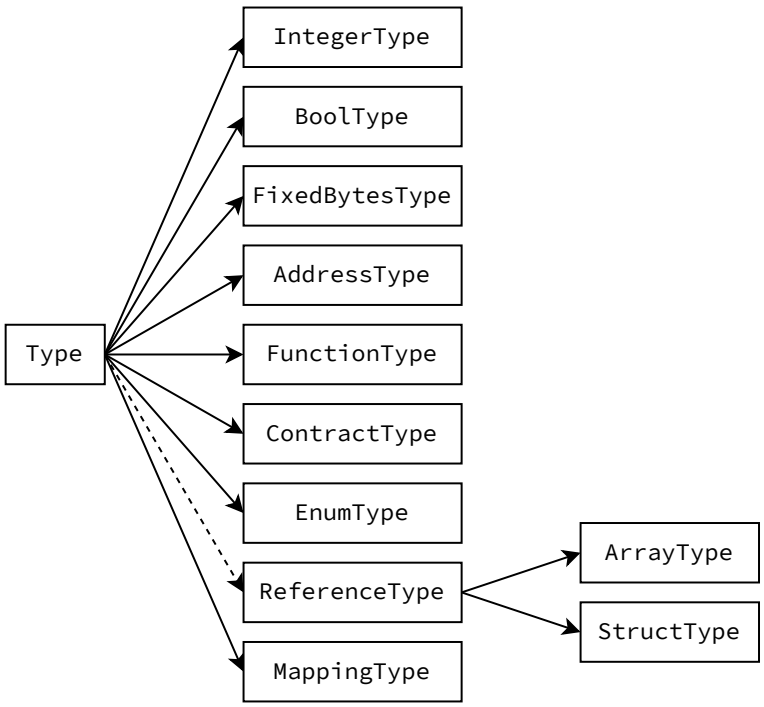


Figure 6.1: `Type` class hierarchy

As shown in Figure 6.1:

- The `ReferenceType` class is abstract. It is annotated with a data location and a boolean flag indicating whether the type is a rebindable pointer or a bound reference. A bound reference is valid only for `storage`, as discussed in Section 5.3.1.
- `MappingType` does not inherit from `ReferenceType`, as explained in Section 5.3.1.

Implementation Examples

Reference Type

```
enum class DataLocation { Storage, Memory, Calldata };

class ReferenceType : public Type {
 public:
  ReferenceType(DataLocation location, bool is_pointer)
      : location(location), is_pointer(is_pointer) {
    assert(is_pointer || location == DataLocation::Storage);
  }

 private:
  DataLocation location;
  bool is_pointer;
};
```

Array Type

```
class ArrayType : public ReferenceType {
 public:
  enum class Kind { Ordinary, Bytes, String };

  // constructor for bytes and string
  ArrayType(DataLocation location, bool is_pointer, bool is_string)
      : ReferenceType(location, is_pointer),
        kind(is_string ? Kind::String : Kind::Bytes) { }

  // constructor for ordinary static arrays
  ArrayType(DataLocation location, bool is_pointer, Type *element, unsigned length)
      : ReferenceType(location, is_pointer),
        kind(Kind::Ordinary), element(element), length(length) { }

  // constructor for ordinary dynamic arrays
  ArrayType(DataLocation location, bool is_pointer, Type *element)
      : ReferenceType(location, is_pointer),
        kind(Kind::Ordinary), element(element) { }

  const Type &getElement() const {
    assert(this->kind == Kind::Ordinary);
    return *this->element;
  }

  // ...

 private:
  Kind kind;
  // for ordinary arrays only
  Type *element;
  // for ordinary arrays only
  std::optional<unsigned> length;
};
```

6.4 Context

All generation procedures receive a reference to a `Context` object, which holds context-related information, such as:

- The current function definition: If the generator is inside a function body, this points to the corresponding `FunctionDefinition`. It is `nullptr` when generating constructs like state variables or inheritance specifiers.
- The current contract definition: This is always valid.
- The list of top-level contract definitions.
- The current scope.

A new `Context` is created whenever the generator steps out of the current generation context to generate a required dependency. Generation then resumes in the previous context once the dependency is complete.

```
class Context {
  FunctionDefinition *current_function;
  ContractDefinition *current_contract;
  std::vector<ContractDefinition *> contracts;
  Scope *scope;
  // ...
};
```

The context can be queried for information such as whether reading from or writing to `storage` is allowed, or for the set of currently visible symbols. For example, the implementation of `canReadFromStorage` may look as follows:

```
bool Context::canReadFromStorage() const {
  // can't read from storage only in the context of a pure function
  return !(this->current_function != nullptr
           && this->current_function->getStateMutability() == StateMutability::Pure);
}
```

6.5 Generation Methods

6.5.1 Types, Expressions, and Statements

The generator uses a unified strategy for generating types, expressions, and statements, as explained in Section 5.2. Each class defines a `static` `generate()` method responsible for producing a node of that type. The method signatures are as follows:

```
static Type *
Type::generate(int d, Context &ctx, const TypeContext &type_ctx);
```

```
static Expression *
Expression::generate(int d, const Type &type,
                     Context &ctx, const ExprContext &expr_ctx);
```

```
static Statement *
Statement::generate(int d, Context &ctx, const StmtContext &stmt_ctx);
```

All three methods share similar signature patterns:

- `int d`: Current node depth in the tree. Used to prevent unbounded recursion, as all three methods are recursive.

- `Context &ctx`: The context object.
- `TypeContext`: Describes the context in which the generated type is used, as described in Section 5.3.2.
- `ExprContext` and `StmtContext`: Provide additional constraints specific to each generation category (e.g., whether the expression must be an l-value).
- In the case of expression generation, the `const Type &type` argument specifies the type that the expression must resolve to.
- Each method returns the corresponding node if generation was successful; otherwise, it returns `nullptr`.

Subclasses implement the corresponding generation methods, which are utilized by the main generation methods. For example:

```
static BinaryExpression *
BinaryExpression::generate(int d, const Type &type,
                           Context &ctx, const ExprContext &expr_ctx);
```

```
static IfStatement *
IfStatement::generate(int d, Context &ctx, const StmtContext &stmt_ctx);
```

In addition to `Type::generate()`, the base `Type` class declares several `virtual` methods for type-directed generation. These must be implemented by all concrete type classes. Examples of such methods include:

Implicit Conversion Source Generation

```
Type *
Type::generateImplicitConversionSource(const Context &ctx,
                                       const TypeContext &type_ctx) const = 0;
```

This method implements the process described in Section 5.4.3. It returns a source type that is implicitly convertible to `*this`.

Type Name Construction

```
TypeName *
Type::toTypeName(const ContractDefinition &scope) const = 0;
```

Returns a `TypeName` AST node specifying this type within the given contract scope.

Binary Operator Operand Generation

```
std::pair<Type *, Type *>
Type::getBinaryOperatorOperands(BinaryOperator op) const = 0;
```

Returns a pair of operand types such that applying `op` to them results in a value of this type.

6.5.2 Source Unit and Declarations

The entry point of generation is the `static` `generate` method of the `SourceUnit` class, which receives the Solidity version to generate for as a parameter:

```
static SourceUnit SourceUnit::generate(Version version);
```

For the process described in Section 5.7, the following methods are required, which are utilized by the `SourceUnit::generate` method:

Empty Contract Generation

```
static void SourceUnit::generateEmptyContracts(Context &ctx);
```

This method creates a list of empty contract definitions and adds them to the context.

Inheritance and Empty Contract Body Elements

```
static void SourceUnit::generateInheritanceAndEmptyContractBodyElements(Context &ctx);
```

This method iterates over the previously created empty contracts and, for each contract:

- Determines the base contracts that the contract will specify in its inheritance specifier list, and also which of these contracts will be provided with arguments directly.
- Creates a list of empty contract body elements.

Struct Member Generation

```
static void SourceUnit::generateStructMembers(Context &ctx);
```

This method traverses the AST and populates struct definitions with members.

Contract Body Element Header Generation

```
static void SourceUnit::generateHeaders(Context &ctx);
```

This method traverses the AST and completes the headers of a contract's body elements, which include the generation of the parameter types for all required declarations (functions, modifiers, events, etc.), or the generation of the type of a state variable.

Completing Contract Generation

```
static void SourceUnit::generateContent(Context &ctx);
```

This method traverses the AST and generates the content of a contract, which includes generating inheritance specifier arguments, the bodies of functions, constructors, modifiers, etc., and the initializer expression of a state variable.

6.5.3 Implementation Examples

The following examples show how various generation methods might be implemented, closely following the logic explained in previous sections of the Generation chapter.

Array Type Generation

```
ArrayType *
ArrayType::generate(int d, Context &ctx, const TypeContext &type_ctx) {
  // select array kind:
  Set<ArrayType::Kind> kinds{ArrayType::Kind::Bytes, ArrayType::Kind::String};
  if (!type_ctx.isMappingKey() && d != MAX_TYPE_DEPTH)
    kinds.insert(ArrayType::Kind::Ordinary);
  ArrayType::Kind kind = kinds.getRandom();

  // select data location and bound reference or rebindable pointer:
  DataLocation location;
  bool is_pointer;
  if (type_ctx.isArrayElement()) {
    location = type_ctx.getArrayDataLocation();
    is_pointer = (location != DataLocation::Storage);
  } else if (type_ctx.isMappingKey()) {
    location = DataLocation::Memory;
    is_pointer = true;
  } else if (type_ctx.isMappingValue()) {
    location = DataLocation::Storage;
    is_pointer = false;
  } else if (type_ctx.isFunctionParameterOrReturnType()) {
    if (type_ctx.getCallConvention() == CallConvention::Internal) {
      location = random<DataLocation>({
        DataLocation::Storage, DataLocation::Memory, DataLocation::Calldata
      });
    } else {
      location = DataLocation::Memory;
    }
    is_pointer = true;
  } else {
    assert(false && "unknown type context");
  }

  if (kind != ArrayType::Kind::Ordinary)
    return new ArrayType(location, is_pointer, kind == ArrayType::Kind::String);

  // generate element type:
  TypeContext element_ctx = TypeContext::makeArrayElement(location);
  Type *element = Type::generate(d + 1, ctx, element_ctx);

  // select static or dynamic:
  bool is_static = flipCoin(0.5);

  if (is_static) {
    unsigned length = random(1, MAX_ARRAY_LENGTH);
    return new ArrayType(location, is_pointer, element, length);
  } else {
    return new ArrayType(location, is_pointer, element);
  }
}
```

Ordinary Assignment Expression Generation

This method is utilized by `AssignmentExpression::generate`.

```
AssignmentExpression *
AssignmentExpression::generateOrdinary(int d, const Type &type,
                                       Context &ctx, ExprContext &expr_ctx) {
  if (d == MAX_EXPR_DEPTH)
    return nullptr;

  if (expr_ctx.lvalueRequired())
    return nullptr;

  if (type.isBoundStorageReference()) {
    if (type.containsMapping())
      return nullptr;
    if (!ctx.canWriteToStorage())
      return nullptr;
  }

  ExprContext left_ctx;
  left_ctx.setLvalueRequired();
  Expression *left = Expression::generate(d + 1, type, ctx, left_ctx);

  TypeContext right_type_ctx = TypeContext::makeExpression();
  Type *right_type = type.generateImplicitConversionSource(ctx, right_type_ctx);
  Expression *right = Expression::generate(d + 1, *right_type, ctx, ExprContext());

  return new AssignmentExpression(AssignmentOperator::Assign, left, right);
}
```

Supporting method: `containsMapping`

```
bool Type::containsMapping() const = 0;
```

This method returns whether this type contains a mapping, in one of the two ways described in Section 5.4.1. For example, the implementation for array types:

```
bool ArrayType::containsMapping() const {
  if (this->kind != Kind::Ordinary)
    return false;
  return this->getElement().isMapping() || this->getElement().containsMapping();
}
```

This method will return `true` for the type:

`mapping(bytes32 => bool)[10] storage reference[] storage reference`

But will return `false` for the following type, as it does not contain a mapping in a way that would require its copying in an assignment.

`function (mapping(uint8 => int8)) internal view[] storage reference`

Array Type Implicit Conversion Source Generation

```
Type *
ArrayType::generateImplicitConversionSource(Context &ctx,
                                            const TypeContext &type_ctx) const {
  if (this->isBoundStorageReference()) {
    DataLocation location;
    bool is_pointer;

    if (type_ctx.isArrayElement()) {
      location = type_ctx.getArrayDataLocation();
      is_pointer = (location != DataLocation::Storage);
    } else {
      location = random<DataLocation>({
        DataLocation::Storage, DataLocation::Memory, DataLocation::Calldata
      });
      is_pointer = (location != DataLocation::Storage || flipCoin(0.5));
    }

    TypeContext element_ctx = TypeContext::makeArrayElement(location);
    Type *element
        = this->getElement().generateImplicitConversionSource(ctx, element_ctx);

    bool is_static = this->isStatic() || flipCoin(0.5);

    if (is_static) {
      unsigned length = random(1, (this->isStatic()
                                   ? this->getLength()
                                   : MAX_ARRAY_LENGTH));
      return new ArrayType(location, is_pointer, element, length);
    } else {
      return new ArrayType(location, is_pointer, element);
    }
  }

  if (this->isRebindableStoragePointer()) {
    // It should be allowed to use either a rebindable pointer or a bound
    // reference without restrictions, as neither case reads from storage.
    bool is_pointer = !ctx.canReadFromStorage() || flipCoin(0.5);
    return this->copyToLocation(DataLocation::Storage, is_pointer);
  }

  if (this->getDataLocation() == DataLocation::Memory) {
    DataLocation location;
    bool is_pointer;

    if (type_ctx.isArrayElement()) {
      location = type_ctx.getArrayDataLocation();
      is_pointer = (location != DataLocation::Storage);
    } else {
      Set<DataLocation> candidate_locations({
        DataLocation::Memory, DataLocation::Calldata
      });
      if (ctx.canReadFromStorage())
        candidate_locations.insert(DataLocation::Storage);
      location = candidate_locations.getRandom();
      is_pointer = (location != DataLocation::Storage || flipCoin(0.5));
    }

    return this->copyToLocation(location, is_pointer);
  }

  if (this->getDataLocation() == DataLocation::Calldata) {
    return this->clone();
  }

  assert(false && "unreachable");
}
```

In contrast to the process described in Section 5.4.3, the code additionally considers the type context of the generated type. Since this process is recursive, the generated type may be used in one of two contexts:

- As the standalone type of an expression (i.e., not nested within another type), or
- As an array element for an array of a known data location.

This distinction is necessary to prevent nested implicit conversions from producing ill-formed types in the context in which they are used.

Supporting Method: `copyToLocation`

```
ReferenceType *
ReferenceType::copyToLocation(DataLocation location, bool is_pointer) const = 0;
```

This method recursively copies the reference type to a new location. It is used, for instance, to "relocate" a `memory` array to `storage` or `calldata`. For example, if:

`T1 = int[] memory[] memory`

Then:

```
T1.copyToLocation(DataLocation::Storage, /*is_pointer=*/true)
```

produces:

`int[] storage reference[] storage pointer`

Chapter 7

Evaluation

In this chapter, we evaluate SolGen in terms of its code coverage and its effectiveness in uncovering compiler bugs. Section 7.1 reports coverage, and Section 7.2 presents discovered bugs.

7.1 Code Coverage

Code coverage is a common metric used to evaluate how thoroughly a test suite exercises a software application's source code. While it does not guarantee the absence of bugs, it serves as a general indicator of test quality and thoroughness. Coverage is typically expressed as a percentage of tested items over the total, and while there is no ideal code coverage number that universally applies to all software, Google provides general guidelines: 60% as *acceptable*, 75% as *commendable*, and 90% as *exemplary* [32]. Common types of coverage metrics include line, function, and branch coverage, which assess the execution of code lines, functions, and decision branches, respectively. Path coverage, which measures the proportion of all possible execution paths tested, is theoretically the most comprehensive metric. However, it is generally impractical, as loops and recursion introduce an infinite number of possible paths, and even when loop iterations are bounded, the total number of paths remains infeasibly large.

7.1.1 Setup

We built solc version 0.5.0 with coverage instrumentation enabled. To collect coverage data, we compiled 1000 SolGen-generated test cases using the default invocation `solc testcase.sol` without any additional compiler flags. Since the front-end is always executed, this is sufficient to evaluate its coverage. Back-end components, which require explicit flags, were not the focus of this analysis. We used lcov[1] and genhtml (which is part of lcov) to generate coverage reports.

We are interested in the coverage of the compiler's front-end, which includes lexical

[1] https://github.com/linux-test-project/lcov

analysis, syntactic analysis, and semantic analysis. This portion of the compiler is illustrated in Figure 7.1, shown within the dotted lines.

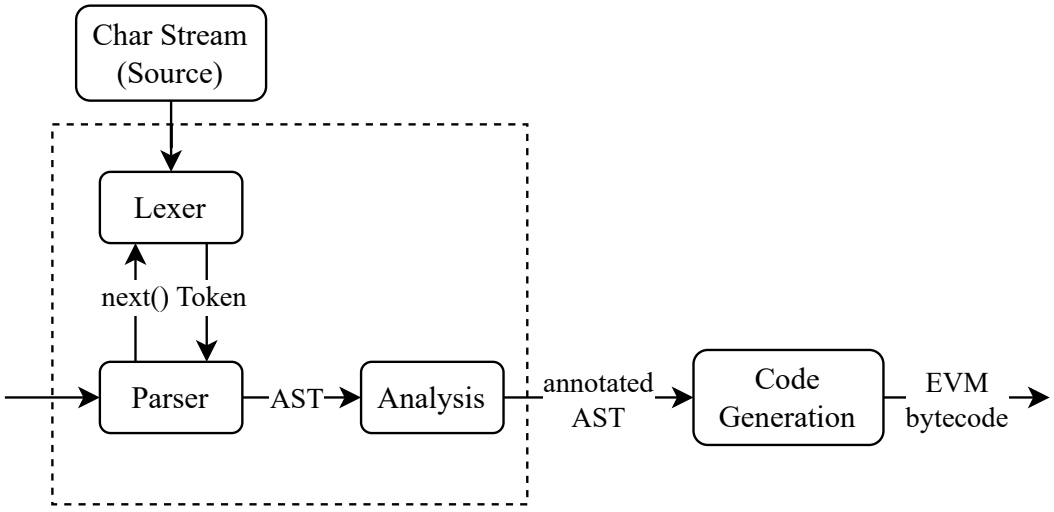


Figure 7.1: Coverage-measured parts

Lexer

Files analyzed:

From `libsolidity/parsing`: `Scanner.cpp` / `h`

	Total	Hit	Coverage
Lines	559	371	66.4%
Functions	74	60	81.1%
Branches	708	292	41.2%

Table 7.1: Coverage of the lexer

Parser

Files analyzed:

From `libsolidity/parsing`: `Parser.cpp` / `h`, `ParserBase.cpp` / `h`

	Total	Hit	Coverage
Lines	1148	972	84.7%
Functions	143	133	93.0%
Branches	2716	992	36.5%

Table 7.2: Coverage of the parser

Semantic Analyzer

Files analyzed:

From `libsolidity/ast`: `ASTAnnotations.cpp` / `h`, `ASTVisitor.h`, `AST_accept.h`, `Types.cpp` / `h`

From `libsolidity/analysis`: all except `DocStringAnalyser.cpp` / `h`, which handles analysis of documentation strings, and `SemVerHandler.cpp` / `h`, which handles semantic versioning.

	Total	Hit	Coverage
Lines	6312	4186	66.3%
Functions	1215	926	76.2%
Branches	14463	4103	28.4%

Table 7.3: Coverage of the semantic analyzer

Discussion

Among the three components, the parser achieves the highest coverage, followed by the lexer, and then the semantic analyzer.

Lexical and syntactic coverage are generally easier to achieve than semantic coverage. A single syntax rule may have multiple semantic interpretations based on context. For example, the production `<expr> → <identifier-expr>` may be covered when referencing a unique declaration (such as a variable or function), but not for cases involving overloaded functions, which require separate semantic handling. Similarly, the production `<expr> → <expr> '=' <expr>` can represent various kinds of assignments depending on operand types, each with distinct semantic behavior.

Interestingly, the parser achieves higher coverage than the lexer. This may seem counterintuitive, but is explainable: lexical analysis includes handling of non-semantic elements such as comments, documentation comments, Unicode literals, and escape sequences. For example, although not difficult to implement, SolGen currently does not generate variants such as `0x01`; it only emits decimal literals (e.g., `1`). These omissions impact the lexer's coverage, but have limited semantic importance.

Line and function coverage are significantly higher than branch coverage. This is expected—branch coverage is more challenging due to control flow complexity. Moreover, the compiler front-end includes many branches that represent invalid execution paths, which SolGen does not attempt to generate. Such branches are, by definition, never taken.

7.1.2 Coverage Saturation Point

To determine how many test cases are needed to reach near-maximal coverage, we measured the incremental increase in coverage for the semantic analyzer as the number of test cases increased from 100 to 1000. This component is the most complex among the three, and its coverage grows more gradually. In contrast, the lexer and parser reach their maximum coverage relatively early—often within the first few hundred test cases—and are therefore excluded from this analysis.

Figures 7.2, 7.3, and 7.4 illustrate the evolution of line, function, and branch coverage in the semantic analyzer, based on the number of test cases used.

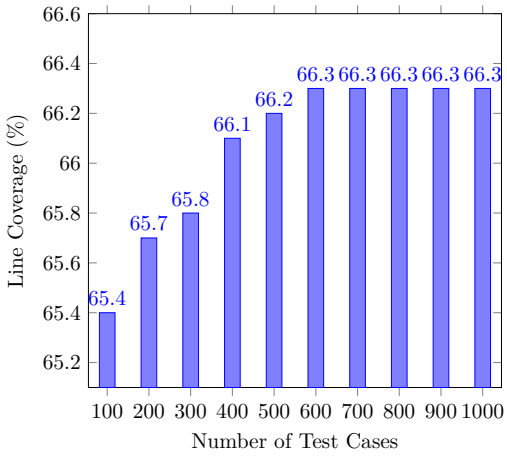


Figure 7.2: Incremental line coverage of the semantic analyzer

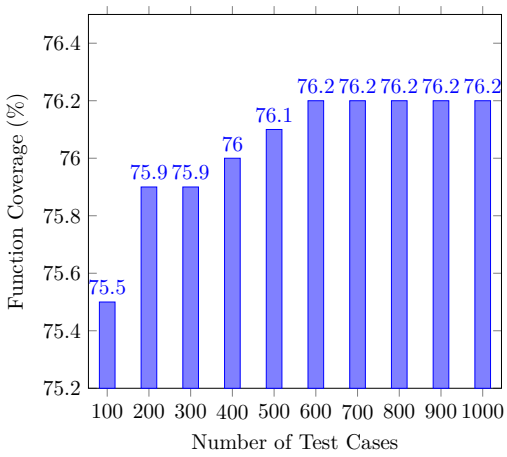


Figure 7.3: Incremental function coverage of the semantic analyzer

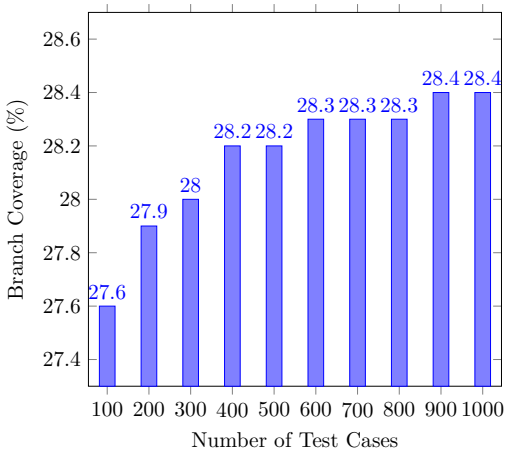


Figure 7.4: Incremental branch coverage of the semantic analyzer

We observe that the rate of coverage improvement slows down significantly after around 600 test cases, with only minor gains beyond that point. By the time we reach 1000 test cases, the coverage levels plateau at approximately:

- 66.3% for lines,
- 76.2% for functions, and
- 28.4% for branches.

Therefore, we consider 1000 test cases sufficient for capturing the majority of coverage possible with the current generator.

7.1.3 Comparison with the compiler's test suite

The Solidity compiler's test suite includes a variety of tests, including a large set of syntax tests. These syntax tests are not designed to be executed; instead, they focus on checking whether the compiler correctly accepts or rejects source code based on its syntactic and semantic validity. Importantly, they include both valid and invalid test cases. For our analysis, we consider only the valid ones—that is, those that the compiler is expected to accept. These test cases were extracted using a Python script and are typically small and focused, each targeting a specific language feature such as implicit type conversions or operations on particular types.

Example Test Cases

`implicit_conversion_from_storage_array_ref.sol`

```
contract C {
  int[10] x;
  int[] y;
  function f() public {
    y = x;
  }
}
```

`index_access_for_bytes.sol`

```
contract C {
  bytes20 x;

  function f(bytes16 b) public view {
    b[uint8(x[2])];
  }
}
```

`external_function_type_public_variable.sol`

```
contract C {
  function (uint) external public x;

  function g(uint) public {
    x = this.g;
  }
  function f() public view returns (function (uint) external) {
    return this.x();
  }
}
```

It should be noted that this test suite is not intended to provide comprehensive standalone coverage, but rather to provide regression testing for known behavior.

To evaluate how SolGen complements the existing test suite, we compiled all valid compiler test cases using the instrumented compiler and extracted their corresponding coverage reports.

The goal of the following comparison is to determine whether SolGen is capable of generating code constructs that the compiler's own tests do not cover, and vice versa. To do this systematically, we perform a set-based comparison of the coverage data.

Set-Based Comparison

From the lcov coverage reports, we extracted:

- `generator_covered`: Set of elements executed by SolGen-generated tests
- `compiler_covered`: Set of elements executed by the compiler's test suite
- `all_instrumented`: Set of all elements tracked by coverage instrumentation

Using these, we define the following:

- `executed_by_both` = `generator_covered` $\cap$ `compiler_covered`
- `only_in_generator` = `generator_covered` $\setminus$ `compiler_covered`
- `only_in_compiler` = `compiler_covered` $\setminus$ `generator_covered`

- `not_covered_by_any` = `all_instrumented` − (`generator_covered` ∪ `compiler_covered`)

Our primary interest lies in the `only_in_generator` and `only_in_compiler` sets. These represent code elements that are uniquely covered by one source but not the other, and therefore directly measure the complementarity between SolGen and the existing test suite. If `only_in_generator` is non-trivial, it indicates that SolGen is capable of generating constructs that exercise parts of the compiler otherwise untouched by manual tests—a key goal of this evaluation.

Lexer

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	559	371 (66.4%)	447 (80.0%)	13 (2.3%)	89 (15.9%)
Functions	74	60 (81.1%)	66 (89.2%)	0 (0.0%)	6 (8.1%)
Branches	708	292 (41.2%)	362 (51.1%)	15 (2.1%)	85 (12.0%)

Table 7.4: Lexer coverage between SolGen and the compiler's test suite

Parser

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	1148	972 (84.7%)	997 (86.8%)	2 (0.2%)	27 (2.4%)
Functions	143	133 (93.0%)	136 (95.1%)	0 (0.0%)	3 (2.1%)
Branches	2716	992 (36.5%)	1022 (37.6%)	7 (0.3%)	37 (1.4%)

Table 7.5: Parser coverage between SolGen and the compiler's test suite

Semantic Analyzer

	Total	Hit (generator)	Hit (compiler)	Hit (only generator)	Hit (only compiler)
Lines	6312	4186 (66.3%)	4860 (77.0%)	97 (1.5%)	771 (12.2%)
Functions	1215	926 (76.2%)	1009 (83.0%)	14 (1.2%)	97 (8.0%)
Branches	14463	4103 (28.4%)	5161 (35.7%)	210 (1.5%)	1286 (8.8%)

Table 7.6: Semantic analyzer coverage between SolGen and the compiler's test suite

The compiler's test suite achieves broader coverage overall across lines, functions, and branches. Our generator adds some unique coverage, but this exclusive coverage remains

limited.

This reflects the current limitations of our generator—mostly due to missing language features—rather than random program generation in general. While it provides some complementary value, most code paths are already exercised by the compiler's tests. Improving the generator could help reveal more uncovered areas.

However, as we will discuss in the bugs section, coverage metrics may not fully capture the generator's contribution to bug finding, where variations in *path coverage*—exploring different input scenarios along the same code paths—can still be valuable.

7.2 Bugs Discovered

SolGen was able to identify a number of bugs throughout all solc versions starting from version 0.5.0, 18 of which were found in solc versions $\geq$ 0.7.0. Table 7.7 shows all previously-unknown bugs that we reported. Most of these are regression bugs introduced in the latest versions of solc. In our reports, we provided a minimal reproducible example (MRE) that triggers the bug rather than the SolGen-generated test cases. Other bugs we encountered were already known to the Solidity developers, most of which have been fixed.

Bug Category	URL	Status
Invalid Diagnostic	https://github.com/argotorg/solidity/issues/14624	fixed
ICE	https://github.com/argotorg/solidity/issues/14792	fixed
ICE	https://github.com/argotorg/solidity/issues/14929	fixed
ICE	https://github.com/argotorg/solidity/issues/14959	fixed
ICE	https://github.com/argotorg/solidity/issues/15308	fixed
ICE	https://github.com/argotorg/solidity/issues/16223	pending
ICE	https://github.com/argotorg/solidity/issues/16225	pending
Segfault	https://github.com/argotorg/solidity/issues/16226	pending

Table 7.7: References of discovered previously-unknown bugs

Bug in the Parser. Since Solidity 0.5.0, the unary plus operator has been disallowed. This restriction was originally enforced during analysis, but starting in version 0.8.20, it was moved to the parsing stage. However, the new check was incorrectly implemented, causing valid syntax to be rejected. In the example below, the parser misinterprets the `+` as unary instead of binary when the expression is used as a standalone statement and rejects it. This bug was fixed in version 0.8.27 (see PR #15315).

```
function f() pure {
  (0) + 1;
}
```

Bug in the Type Checker. Solidity 0.8.8 introduced user-defined value types as zero-cost abstractions over elementary types, improving type safety and readability. Solidity also allows forward references to file-level symbols. When arrays or structs containing user-defined value types were used before the type's declaration, as in the example below, an ICE was triggered. This bug was fixed in version 0.8.10 (see PR #12186).

```
contract C {
  T[] v;
}

type T is bool;
```

Bug in the SMTChecker. solc includes a formal verification module called the SMTChecker, which must be explicitly enabled. When analyzing the program below using the CHC (Constrained Horn Clauses) engine, it triggered an ICE. The crash occurs when the operands of a ternary operator are expressions of the empty tuple type—either as empty tuple literals, as shown here, or more complex expressions. This bug was fixed in version 0.8.26 (see PR #15025).

```
contract C {
  function f() public pure {
    true ? () : ();
  }
}
```

Bug in the Code Generator. While SolGen cannot verify the correctness of the code generator, some issues are detectable at compile-time. The program below triggers an ICE in solc versions $\leq$ 0.8.1. Here, `f` is a local variable of function type, and it is called with an argument of a function type that is implicitly convertible but not identical—`f` expects a `view` function, but receives a `pure` one. An assertion in the back-end was too strict and failed to account for this case, causing the crash. This bug was fixed in version 0.8.2 (see PR #10959).

```
contract C {
  function h() public view {
    function () external pure g;
    function (function () external view) external pure f;
    f(g);
  }
}
```

The test cases generated by SolGen are similar in structure and purpose to the syntactic tests in the Solidity compiler's test suite. The compiler already includes an extensive set of test cases covering both valid and invalid programs. However, our tool provides a way to generate such test cases automatically and at scale, without requiring manual effort or introducing developer bias. This improves efficiency by eliminating the need to write test cases by hand, and increases bug-finding potential, as manually written tests often fail to exhaustively cover all relevant edge cases. For example, the SMTChecker previously failed to handle some assignments between arrays when the element types were implicitly convertible but not identical. Consider the case for function types:

```
contract C {
  function () internal view[] v1;
  function () internal pure[] v2;

  function f() public {
    v1 = v2;
  }
}
```

This was fixed in version 0.8.27 (see PR #15322) for function and contract types, and corresponding regression tests were added. However, SolGen later found that the same issue remained unaddressed for address types:

```
contract C {
  address[] v1;
  address payable[] v2;

  function f() public {
    v1 = v2;
  }
}
```

In this example, the type of `v2` is implicitly convertible to the type of `v1` because `address payable` is implicitly convertible to `address`.

This was fixed later, in version 0.8.28 (see PR #15420). This shows how manually writing test cases can overlook variations of the same core problem. In contrast, a random program generator systematically explores a broader and less biased subset of the input space, increasing the chances of revealing such missed edge cases. As noted earlier with coverage, while the compiler may include test cases for implicit conversions between address types—and this does reflect in coverage reports—it does not cover cases where the conversion occurs in the context of an array's element type, leading to unhandled edge cases. This limitation is not revealed by coverage reports.

During our test run, we noticed a recurring pattern: compiler bugs often arise in newly introduced language features and tend to hide in untested or rarely used parts of the language. Consider the two programs below. Both trigger an ICE in the latest version of solc (0.8.30 as of this writing).

```
contract C {
  function f(S[10] calldata) internal { }
}

struct S {
  function () internal pure m;
}
```

```
type T is bool;

library L {
  function f(T[] storage p) external { }
}
```

In the first program, the issue is related to having static arrays of structs that contain an internal function type in `calldata`. While such types are valid, they are effectively useless in `calldata`: only external calls can populate `calldata`, and external calls cannot pass types that contain internal function types. As a result, this corner case remained untested. Furthermore, both bugs were introduced alongside new language features: the first with `calldata` parameters for internal functions (introduced in version 0.6.9) and the second with user-defined value types (introduced in version 0.8.8). These examples support the broader observation that new features are particularly prone to bugs, especially when their

interactions with less common language constructs are not thoroughly tested.

Regarding the importance of the discovered bugs, while compiler crashes—particularly those triggered by invalid input—are generally not treated as high-priority issues, they still reveal weaknesses in the compiler's reliability. High-priority bugs are typically those that lead to incorrect program behavior, runtime failures, or security vulnerabilities. The bugs we discovered often involve specific edge cases that are unlikely to occur in typical, hand-written code. Nevertheless, the compiler should be able to handle them properly, especially given that the inputs are valid programs. We argue that identifying such issues contributes to the overall quality, reliability, and maintainability of the Solidity compiler's codebase.

Chapter 8

Conclusion and Future Work

In this thesis, we presented SolGen, a generator of semantically correct Solidity programs. We evaluated its effectiveness in terms of compiler code coverage and its ability to uncover bugs. As shown in Chapter 7, the achieved code coverage was decent, but not outstanding, and the generation of currently unsupported language constructs is required. SolGen discovered several bugs—primarily compiler crashes—across versions of solc starting from version 0.5.0. Of these, eight were previously unknown bugs and were reported to the Solidity developers.

8.1 Difficulties and Delimitations

We previously described the generation process as the reverse of what a compiler front-end does. However, in practice, it is often easier for the compiler to check whether certain restrictions apply than it is for a generator to enforce them during generation. One example of this is function overloading. In Solidity, a contract can define multiple functions with the same name as long as their parameter types differ. For `public` and `external` functions, which are part of the contract's interface, their parameter types must differ not only in terms of Solidity types but also in terms of their external (ABI) types. This is shown in the example below:

```
contract C {
  function f(address) private { }
  // this is allowed
  function f(C) internal { }

  function g(address) public { }
  // error
  function g(C) external { }
}
```

Here, `f`'s overloads are valid but `g`'s are not, as both address types and contract types map to address types in the ABI and thus the signatures of these two functions do not differ in the ABI.

The way overload resolution works is by comparing the types of the provided arguments against all overloads to find a list of candidate functions. A function is considered a

candidate if all arguments can be implicitly converted to its parameter types. Resolution succeeds only if exactly one match is found. In the example below, overload resolution fails in the first call because both overloads are valid, as the type of the argument `int8` is implicitly convertible to both `int8` and `int16`. However, overload resolution succeeds in the second call as the argument type `int16` is implicitly convertible only to `int16`.

```
contract C {
  function f(int8) internal pure { }
  function f(int16) internal pure { }

  function g() public pure {
    // error
    f(int8(127));

    // this is allowed
    f(int16(128));
  }
}
```

This is straightforward for the compiler to implement as a type-checking step. The reverse process—generating argument types that are implicitly convertible to exactly one overload and not to any others—is significantly harder in the general case, especially when conversion rules overlap. This reflects a broader asymmetry: while a compiler can verify multiple constraints independently and incrementally, a generator must satisfy all of them simultaneously during synthesis.

A related example lies in the compiler's ability to perform multiple passes over the AST, each dedicated to a distinct concern. For instance, the Solidity compiler performs type checking in one traversal (via the `TypeChecker` visitor) and enforces `pure`/`view` restrictions in a separate pass (via the `ViewPureChecker` visitor). In contrast, a generator must handle type correctness and mutability constraints in a single forward pass.

A delimitation of this generator is its inability to generate programs that are free of undefined behavior and independent of unspecified behavior at runtime. As a result, even if solc miscompiles a generated program, we are unable to automatically detect this—for example, via differential testing. Generating such programs, especially for Solidity, is not a trivial task and was left outside the scope of this thesis. A common problem is avoiding unspecified behavior in the order of evaluation of expressions [33], such as with the arguments of a binary operator or a function call, where complex code-generation-time approaches would be required. Figure 8.1 shows a simple example of unspecified behavior in a Solidity program that we would need to avoid.

```
contract C {
  function f() public pure returns (int) {
    int x = 0;
    (x = 1) + (x = 2);
    return x;  // value of x is implementation-dependent
  }
}
```

Figure 8.1: Example of unspecified behavior in a Solidity program

8.2 Future Work

Our goal is to focus on the latest released Solidity version and extend SolGen to generate currently unsupported language constructs, in order to increase compiler code coverage and potentially uncover previously unknown bugs. The main language constructs currently unimplemented are:

- function overriding
- function overloading
- inline assembly
- fixed-point numbers and types, mainly because they are mostly unimplemented by solc itself.

The process described in Section 5.7 currently supports only unique identifiers and does not account for more complex symbol resolution features such as function overriding, function overloading, or symbol shadowing. To support these features, the symbol generation mechanism must be extended.

For example, when creating a new symbol in a given scope, if that symbol corresponds to a function definition, we may choose to reuse an existing function name from the current scope to create an overloaded version. This would require either changing the number of parameters or, if the parameter count remains the same, ensuring that at least one parameter type differs between the two signatures. Supporting this behavior in the general case (for an arbitrary number of overloads, not just two) necessitates improvements to the current `Type::generate` method, as its design becomes increasingly unsustainable when type generation must respect such context-sensitive constraints.

A more scalable alternative that is currently under development is the introduction of a type pool. Types are generated bottom-up: we begin by inserting simple leaf types (of depth 0) such as `int32`, `bool`, `address`, and `bytes memory` into the pool. These types are then used to construct composite types, which are recursively added back into the pool. For instance, we might create types like `bytes memory[] memory`, `mapping(int32 => bool)`, or `address[42] storage reference`, and use these as building blocks for more complex types. This approach enables the construction of a reusable pool of types, which can be filtered based on constraints at the point of use. This is a significantly cleaner and more flexible alternative to the current `Type::generate` method.

Another area in need of improvement is the generation of expressions. At present, the type passed to `Expression::generate` is generated without considering the broader context, leading to a high rate of discarded or unusable expressions. This results in unused declarations—particularly for function parameters and return parameters—which are fixed and cannot be modified without breaking consistency elsewhere. While directed generation remains useful in contexts where expression generation **must** succeed (e.g., inheritance specifiers), it should not be the sole mechanism relied upon. A more balanced

and context-aware generation strategy is needed to ensure better utilization of available symbols and types.

Solidity is a relatively new programming language with ongoing changes and new features, such as a more powerful type system currently under development, which leave plenty of room for improvement in the codebase of solc and we hope that SolGen can contribute to this effort.

References

[1] Haoyang Ma. *A Survey of Modern Compiler Fuzzing*. 2023. DOI: 10.48550/ARXIV.2306.06884. URL: https://arxiv.org/abs/2306.06884.

[2] Xavier Leroy. "Formal verification of a realistic compiler." In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: http://dx.doi.org/10.1145/1538788.1538814.

[3] *Solidity Documentation*. Solidity. URL: https://soliditylang.org/ (visited on 09/25/2025).

[4] Alex Groce. *afl-compiler-fuzzer*. GitHub repository. GitHub. URL: https://github.com/agroce/afl-compiler-fuzzer (visited on 07/14/2025).

[5] Solidity Contributors. *Running the fuzzer via AFL*. Solidity Documentation. 2019. URL: https://docs.soliditylang.org/en/v0.5.10/contributing.html#running-the-fuzzer-via-afl (visited on 07/14/2025).

[6] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. "Making no-fuss compiler fuzzing effective." In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. CC '22. ACM, Mar. 2022, pp. 194–204. DOI: 10.1145/3497776.3517765. URL: http://dx.doi.org/10.1145/3497776.3517765.

[7] Charalambos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis, and Dimitris Mitropoulos. "Syntax-Aware Mutation for Testing the Solidity Compiler." In: *Computer Security – ESORICS 2023*. Springer Nature Switzerland, 2024, pp. 327–347. DOI: 10.1007/978-3-031-51479-1_17. URL: http://dx.doi.org/10.1007/978-3-031-51479-1_17.

[8] Solidity Contributors. *An Introduction to Solidity's Fuzz Testing Approach*. Solidity Blog. Feb. 10, 2021. URL: https://soliditylang.org/blog/2021/02/10/an-introduction-to-soliditys-fuzz-testing-approach/ (visited on 07/14/2025).

[9] Arpan Thaman. *How We Test the Compiler Backend*. Microsoft DevBlogs. Sept. 18, 2019. URL: https://devblogs.microsoft.com/cppblog/how-we-test-the-compiler-backend/ (visited on 07/14/2025).

[10] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." In: *Communications of the ACM* 33.12 (Dec. 1990), pp. 32–44. DOI: 10.1145/96267.96279. URL: http://dx.doi.org/10.1145/96267.96279.

[11] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. "A systematic review of fuzzing techniques." In: *Computers & Security* 75 (June 2018), pp. 118–137. DOI: 10.1016/j.cose.2018.02.002. URL: http://dx.doi.org/10.1016/j.cose.2018.02.002.

[12] Wikipedia contributors. *Fuzzing*. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Fuzzing (visited on 07/14/2025).

[13] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. 2020. DOI: 10.48550/ARXIV.2002.12543. URL: https://arxiv.org/abs/2002.12543.

[14] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs." In: *ACM SIGPLAN Notices* 49.6 (June 2014), pp. 216–226. DOI: 10.1145/2666356.2594334. URL: http://dx.doi.org/10.1145/2666356.2594334.

[15] William M. McKeeman. "Differential Testing for Software." In: *Digital Technical Journal* 10.1 (Dec. 1998), pp. 100–107. URL: https://www.cs.swarthmore.edu/~bylvisa1/cs97/f13/Papers/DifferentialTestingForSoftware.pdf (visited on 09/25/2025).

[16] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." In: *ACM SIGPLAN Notices* 46.6 (June 2011), pp. 283–294. DOI: 10.1145/1993316.1993532. URL: http://dx.doi.org/10.1145/1993316.1993532.

[17] *How to submit an LLVM bug report*. LLVM Project. URL: https://llvm.org/docs/HowToSubmitABug.html (visited on 09/25/2025).

[18] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-case reduction for C compiler bugs." In: *ACM SIGPLAN Notices* 47.6 (June 2012), pp. 335–346. DOI: 10.1145/2345156.2254104. URL: http://dx.doi.org/10.1145/2345156.2254104.

[19] Andreas Zeller. "Yesterday, my program worked. Today, it does not. Why?" In: *ACM SIGSOFT Software Engineering Notes* 24.6 (Oct. 1999), pp. 253–267. DOI: 10.1145/318774.318946. URL: http://dx.doi.org/10.1145/318774.318946.

[20] Michal Zalewski. *american fuzzy lop (AFL)*. https://lcamtuf.coredump.cx/afl/. Greybox fuzzer with compile-time instrumentation and coverage-guided mutation. 2014.

[21] Kostya Serebryany. *LibFuzzer: A Library for Coverage-Guided Fuzz Testing*. https://llvm.org/docs/LibFuzzer.html. In-process, coverage-guided fuzzer for LLVM-based targets. 2015.

[22] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments: Finding Bugs in JavaScript Engines." In: *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA, USA: USENIX Association, 2012, pp. 445–458. DOI: 10.5555/2362793.2362831. URL: https://dl.acm.org/doi/10.5555/2362793.2362831.

[23] Richard Rutledge, Sunjae Park, Haider Khan, Alessandro Orso, Milos Prvulovic, and Alenka Zajic. "Zero-Overhead Path Prediction with Progressive Symbolic

Execution." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, May 2019, pp. 234–245. DOI: 10.1109/icse.2019.00039. URL: http://dx.doi.org/10.1109/ICSE.2019.00039.

[24] Vu Le, Chengnian Sun, and Zhendong Su. "Finding deep compiler bugs via guided stochastic program mutation." In: *ACM SIGPLAN Notices* 50.10 (Oct. 2015), pp. 386–399. DOI: 10.1145/2858965.2814319. URL: http://dx.doi.org/10.1145/2858965.2814319.

[25] Chengnian Sun, Vu Le, and Zhendong Su. "Finding compiler bugs via live code mutation." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. SPLASH '16. ACM, Oct. 2016, pp. 849–863. DOI: 10.1145/2983990.2984038. URL: http://dx.doi.org/10.1145/2983990.2984038.

[26] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "Nautilus: Fishing for Deep Bugs with Grammars." In: *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. Internet Society. San Diego, CA, USA, Feb. 2019. URL: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/.

[27] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. "Grammar-based testing for little languages: an experience report with student compilers." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SPLASH '20. ACM, Nov. 2020, pp. 253–269. DOI: 10.1145/3426425.3426946. URL: http://dx.doi.org/10.1145/3426425.3426946.

[28] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars." In: *Empirical Software Engineering* 22.2 (Jan. 2016), pp. 928–961. DOI: 10.1007/s10664-015-9422-4. URL: http://dx.doi.org/10.1007/s10664-015-9422-4.

[29] Sepideh Maleki, Annie Yang, and Martin Burtscher. "Higher-order and tuple-based massively-parallel prefix sums." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. ACM, June 2016, pp. 539–552. DOI: 10.1145/2908080.2908089. URL: http://dx.doi.org/10.1145/2908080.2908089.

[30] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. "Many-core compiler fuzzing." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, June 2015, pp. 65–76. DOI: 10.1145/2737924.2737986. URL: http://dx.doi.org/10.1145/2737924.2737986.

[31] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ICFP00. ACM, Sept. 2000, pp. 268–279. DOI: 10.1145/351240.351266. URL: http://dx.doi.org/10.1145/351240.351266.

[32] Carlos Arguelles, Marko Ivanković, and Adam Bender. *Code Coverage Best Practices*. Google Testing Blog. Aug. 7, 2020. URL: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html (visited on 09/25/2025).

[33] Solidity Documentation. *Expressions and Control Structures: Order of Evaluation of Expressions*. Section "Order of Evaluation of Expressions". URL: https://docs.solidity.org/en/latest/control-structures.html (visited on 09/25/2025).