CS335 Milestone 2 : ProofEngine

Pragati Agrawal (220779), Dhruv Gupta (220361), Kundan Kumar (220568)

19 March 2025

Work done till now

- SMTLIB code generation for the *constraint* file : Parsing the constraint file, converting infix notation to prefix notation, generating `assert` statement for constraint.
- SMTLIB code generation for *assignment* statements : Converting each *assignment* statement to an equivalent `assert` statement.
- SMTLIB code generation for *if-else* blocks : Extracting the *if-else* blocks from the **Control Flow Graph (CFG)** and converting each into equivalent `assert + ite` statement.
- *Variable declarations* in SMTLIB code : Extracting the variable names from the generated SMTLIB code, add variable declarations and `check-sat` statement to get complete SMTLIB code.
- SMTLIB code for command line initialization : Equivalent `assert` statements are added for initialization of variables done using the `-d/--params` flag.
- *Validity checks* for correctness : Checking if source code follows the assumptions we have made (refer to Assumptions). Also a precautionary check to validate the final SMTLIB code is added.

The generated code is dumped into an output file which can be given to a Z3 solver. We designed some test cases to verify our implementation and used 2 online Z3 solvers Z3 Online Demostrator and jfmc's Z3 Playground to check the validity of the code. Some test cases were designed to verify the error handling too.

Overview of modules written/modified

- **chiron.py**: We have added a command line flag `-smt/--smtlib` which expects two arguments – path to constraint file, path to output file where SMTLIB code will be dumped. If this flag is provided, then the code to generate SMTLIB code is invoked.
  - Firstly, we parse the constraint file and generate a corresponding SMTLIB `assert` statement using the **ConstraintToSmtlib.py** module.
  - Then we generate SMTLIB code for the source program using the **CFGToSmtlib.py** module.

  - Then we generate SMTLIB code for the initialization of parameters (given by the `-d/--params` flag).
  - Then we check the sanity of the generated code using the **CheckOutput.py** module.
  - Then we generate SMTLIB code for variable declarations.
  - Then we add `check-sat` and `get-model` SMTLIB statements.
  - Finally we dump the complete code in the output file.

- **ConstraintToSmtlib.py**: It uses the **ConstraintParser.py** module to extract `literal_groups` and `expr_dict`. Then all the expressions in the `expr_dict` are converted from Infix notation to Prefix notation using the **Infix_To_Prefix.py** module. This to generate the equivalent SMTLIB `assert` statement for the constraint in CNF form using **CNF_To_SMTLIB.py** module.

- **ConstraintParser.py**: Parses the constraints file and constructs two key data structures:
  - `expr_dict`: A dictionary mapping literals to their corresponding expressions (stored in infix notation for readability).
  - `literal_groups`: A 2D list representing the CNF expression in terms of literals.
    - Each inner list corresponds to a clause (disjunction of literals).
    - The entire list represents the CNF formula (conjunction of clauses).

  Example:

  $$[\,[\,c, \sim a\,],\ [\,d, \sim e, \sim f\,]\,]$$

  This represents the CNF formula:

  $$(c \vee \neg a) \wedge (d \vee \neg e \vee \neg f)$$

- **Infix_To_Prefix.py**: Converts expressions from **infix notation** to **prefix notation**, as required by SMTLIB.
  - Constructs an **Abstract Syntax Tree (AST)** for the given expression using the **ast** library of python.
  - Performs a **preorder traversal** to generate the prefix notation.

- **CNF_To_SMTLIB.py**: Uses `literal_groups` and `expr_dict` (converted to prefix notation) to generate the equivalent SMTLIB representation of the CNF expression.

- **CFGtoSmtlib.py**: For generating the SMTLIB code for the source code we use the Control Flow Graph (CFG) of Chiron.
  - Firstly, we check the validity of the CFG. This check is to ensure that the source code follows the assumptions on the code that we have taken (refer to Assumptions). The first check we do is that of whether the CFG generated is a Directed Acyclic Graph. This ensures the CFG and hence the source code has no loops. Then we check that there must not be nested *if-else* blocks. For this we check that each node must have degree less than or equal to 2, and if it is two, it means we are on an if node. Then we perform DFS of the CFG to find the convergence node of the if and else paths. If no convergence node can be found, the input source code is not in the required format as assumed. Also, according to *Assumption 5*, there must be exactly one statement in each *if-else* block, and it must assign to the same variable. This check is done in the function **if_else_smtlib** in the module.

  - Then we extract the execution order of the statements from the CFG and generate corresponding SMTLIB `assert` statement for each line. We perform a DFS on the CFG and find the linear code flow in the graph. In case of branches, we maintain tuple of {*if_condition, if_statements, else_statements*}. And after all this, we use the *if_else_smtlib* and *assign_smtlib* functions to write suitable SMTLIB `assert` statements.

  Using the CFG makes it easier to check the assumptions on the source code that we have made. Extracting the execution order primarily decompiles the *if-else* block which can then be converted to a single SMTLIB statement.

- **CheckOutput.py**: After the SMTLIB code is generated for both the constraint file and the source code, we perform some sanity checks to ensure that the SMTLIB code generated is indeed valid.
  - First, we check if any of the literals is replaced by '`None`' in the `assert` statements. It may occur in some cases where literal is invalid or a syntax error has occurred during SMTLIB code generation.
  - Second, we check the set of variables/literals used in the constraint file must be a subset of the the set of variables/literals used in the source code.

  The first check is performed only as a precautionary measure since we have added syntax checks at each point during code generation. The second check is to ensure that the constraint file does not put constraints on any variables/literals not present in the source code. We have performed this check after the code generation since after this code for the constraint file and the program file is in the same format which makes the checking function simpler.

Assumptions

There are some assumptions on the source code that we have made for simplicity:

1. The code only contains *assignment* statements and *if-else* blocks, i.e., there are no loops and no turtle commands (eg. `pendown`, `forward`, `left`, etc.)
2. The code is written in **SSA-like** form, i.e., each variable can be assigned a value only once. We do not need ϕ nodes (full SSA form) for our purpose.
3. Each *if* block has a corresponding *else* block.
4. There are no nested *if-else* blocks.
5. Each *if-else* block must be of the following format:

```
if (condition) [
    var = value1
] else [
    var = value2
]
```

   Note that here the `var` must be same in the *if* and *else* blocks. The `value1` and `value2` can be any arithmetic expressions. The `condition` can be a conjunction or disjunction of multiple comparisons. For eg., `((:x < :y && :x > :z) || :flag == 0)` is a valid `condition`.

6. The set of variables used in the constraint file must be a subset of the set of variables used in the source code and those inititalized using `-d/--params` flag.

7. To maintain SSA-like form it should be the case that if a variable is initialized by `-d/--params` flag, it should not be assigned another value.

8. SMTLIB uses `"ite", "and", "or", "not", "assert", "div"` as keywords. So, they cannot be used as variables in the source code.

There are also a few dependencies on the implementation of Chiron, i.e., we have exploited some implementation choices in Chiron:

1. Each basic block has only one statement. This makes it easier to generate SMTLIB code line by line.

2. Variable names remain same when IR and CFG are generated. This makes it easier to generate SMTLIB code for constraint file. If the intermediate modules changed the source code variable names, then same changes would have to be applied for variables in constraint file.

3. The *if-else* blocks are compiled in CFG such that the *cond_true* branch is the *if* branch and the *cond_false* branch is the *else* branch. This allows us to identify the *if* and *else* branches while decompiling the *if-else* block.

4. Syntax checking of the source code upto CFG generation has been assumed to be correct.

5. Chiron supports only integer datatype. Therefore, we use `Int` datatype in variable declarations in the SMTLIB code.

Format of constraint file

The constraint file should contain two blocks separated by $\backslash n \backslash n$ (i.e., a blank line).

- *First Block* should be a list of literals mapped to corresponding expressions, i.e., each entry should be of the form {`literal:  expression`}. Here `expression` must be a comparative expression with a single operator (e.g., `x < 20`, `y = 5`). Note that we have used = for equality check instead of ==, since there is no need of assignment in constraint. Also, since ! = operator is absent in SMTLIB, it cannot be used. Therefore to write `p:x!=y`, one must write `p:x=y` and use `~p`.

- *Second Block* represents the constraint in **Conjunctive Normal Form (CNF)**. Each line is of the form `literal1, literal2, ...`
  This block is to be interpreted as:

  – For each line, take disjunction (OR) of all the literals. This gives us the clauses in our CNF.
  – Take conjunction (AND) of all the clauses.
  – ~ denotes logical negation (NOT).
  – Replace each literal in the CNF by the corresponding expression using the map defined by the *First Block* to obtain the constraint.

  A typical constraint file may look like this -

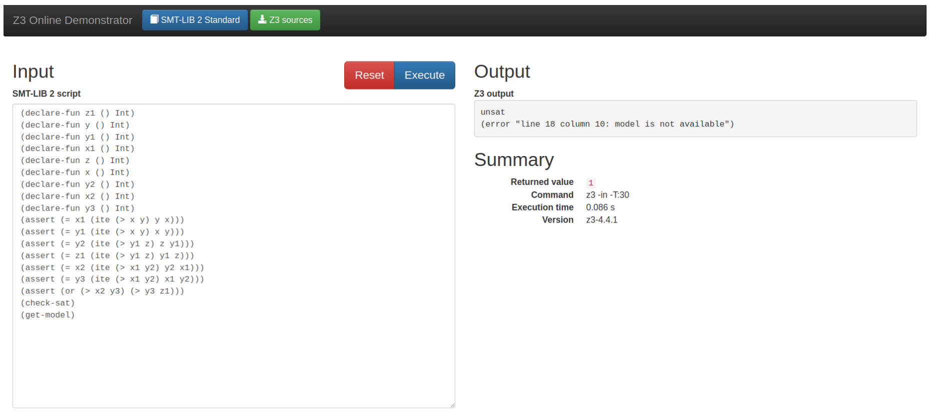
```
p:x<0
q:x>0
r:x=0
a:sign1=0
b:sign1=-1
c:sign1=1

p,q,r
p,q,~a
p,~c,r
p,~c,~a
~b,q,r
~b,q,~a
~b,~c,r
~b,~c,~a
```

Test Cases

All the test cases are written in the **example** folder inside ChironCore. The constraint files for these files is also present in the same folder. The output smtlib code generated is stored in a file at the relative address provided by the user.

- **Sorting three numbers:** The code file **sort.tl** sorts three numbers x,y,z in ascending order and the constraint file checks if these satisy the condition `x <= y <= z`. To check satisfiability, we have negated the constraint and written in the constraint file **cons_sort.txt**, because if the negated constraint is satisfiable, this means our code is incorrect. So the negated constraint would be `(x > y or y > z)`. Figure 1 is the screenshot of the run of its code on Z3 Online Demostrator:



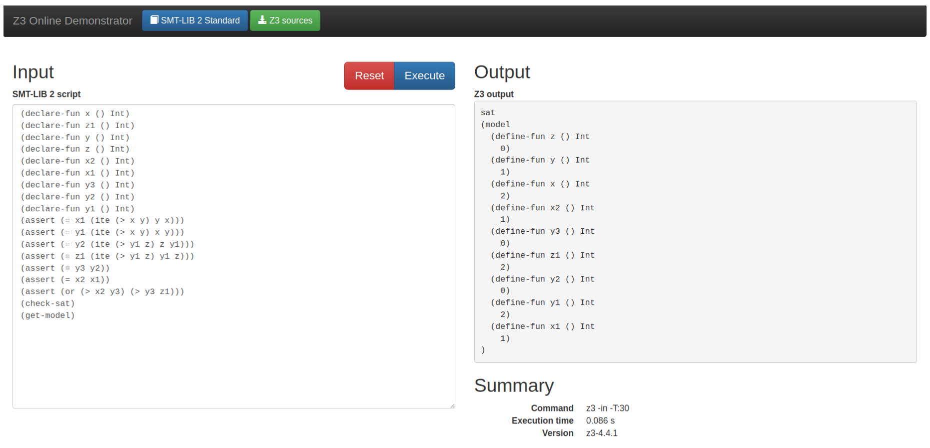
```
(declare-fun z1 () Int)
(declare-fun y () Int)
(declare-fun y1 () Int)
(declare-fun x1 () Int)
(declare-fun z () Int)
(declare-fun x () Int)
(declare-fun y2 () Int)
(declare-fun x2 () Int)
(declare-fun y3 () Int)
(assert (= x1 (ite (> x y) y x)))
(assert (= y1 (ite (> x y) x y)))
(assert (= y2 (ite (> y1 z) z y1)))
(assert (= z1 (ite (> y1 z) y1 z)))
(assert (= x2 (ite (> x1 y2) y2 x1)))
(assert (= y3 (ite (> x1 y2) x1 y2)))
(assert (or (> x2 y3) (> y3 z1)))
(check-sat)
(get-model)
```

```
unsat
(error "line 18 column 10: model is not available")
```

```
 Returned value   1
        Command   z3 -in -T:30
 Execution time   0.086 s
        Version   z3-4.4.1
```

Figure 1: Output of sort.tl

- **Sorting three numbers (buggy):** The code file **sort_buggy.tl** tries to sorts three numbers x,y,z in ascending order but it has a bug in it. The bug is that if the inputs are in descending order (i.e. if $x > y > z$) then the code returns the order (y, z, x) which is incorrect. The constraint file remains the same as for sort.tl but this time our constraint should be satisfiable and z3 should also give us a set of values of variables where it fails. Figure 2 is the screenshot of the run of its code on Z3 Online Demostrator:



SMT-LIB 2 script

```
(declare-fun x () Int)
(declare-fun z1 () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(declare-fun x2 () Int)
(declare-fun x1 () Int)
(declare-fun y3 () Int)
(declare-fun y2 () Int)
(declare-fun y1 () Int)
(assert (= x1 (ite (> x y) y x)))
(assert (= y1 (ite (> x y) x y)))
(assert (= y2 (ite (> y1 z) z y1)))
(assert (= z1 (ite (> y1 z) y1 z)))
(assert (= y3 y2))
(assert (= x2 x1))
(assert (or (> x2 y3) (> y3 z1)))
(check-sat)
(get-model)
```

Z3 output

```
sat
(model
  (define-fun z () Int
    0)
  (define-fun y () Int
    1)
  (define-fun x () Int
    2)
  (define-fun x2 () Int
    1)
  (define-fun y3 () Int
    0)
  (define-fun z1 () Int
    2)
  (define-fun y2 () Int
    0)
  (define-fun y1 () Int
    2)
  (define-fun x1 () Int
    1)
)
```

Summary

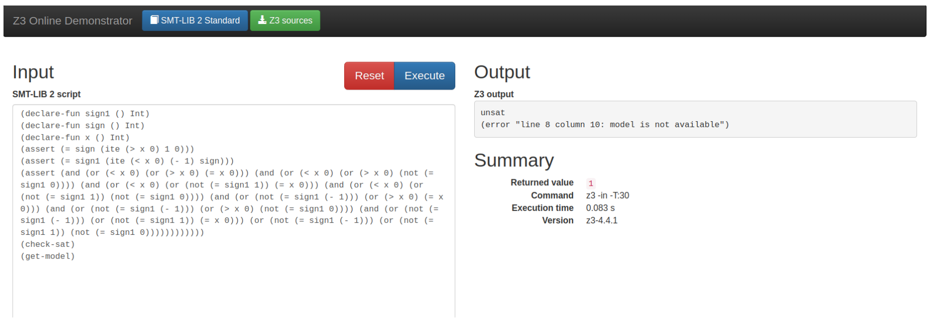
Command	z3 -in -T:30
Execution time	0.086 s
Version	z3-4.4.1

Figure 2: Output of sort_buggy.tl

- **Signum Function:** The code **signum.tl** performs the signum function which is defined as:

$$\text{signum}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \\ 1, & \text{if } x > 0 \end{cases}$$

The constraint file **cons_signum.txt** checks the negated version of the condition for the signum function, i.e. `(x > 0 and sign = 1) or (x < 0 and sign = -1) or (x = 0 and sign = 0)`. Figure 3 is the screenshot of the run of its code on Z3 Online Demostrator:



SMT-LIB 2 script

```
(declare-fun sign1 () Int)
(declare-fun sign () Int)
(declare-fun x () Int)
(assert (= sign (ite (> x 0) 1 0)))
(assert (= sign1 (ite (< x 0) (- 1) sign)))
(assert (and (or (< x 0) (or (> x 0) (= x 0))) (and (or (< x 0) (or (> x 0) (not (=
sign1 0)))) (and (or (< x 0) (or (not (= sign1 1)) (= x 0))) (and (or (< x 0) (or
(not (= sign1 1)) (not (= sign1 0)))) (and (or (not (= sign1 (- 1))) (or (> x 0) (= x
0))) (and (or (not (= sign1 (- 1))) (or (> x 0) (not (= sign1 0)))) (and (or (not (=
sign1 (- 1))) (or (not (= sign1 1)) (= x 0))) (or (not (= sign1 (- 1))) (or (not (=
sign1 1)) (not (= sign1 0))))))))))))
(check-sat)
(get-model)
```

Z3 output

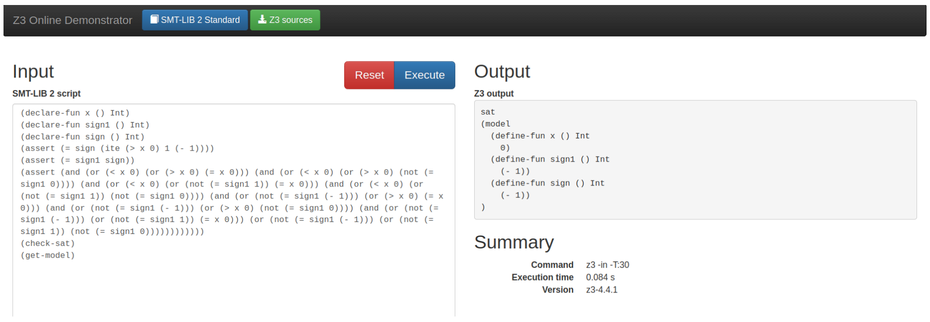
```
unsat
(error "line 8 column 10: model is not available")
```

Summary

Returned value	1
Command	z3 -in -T:30
Execution time	0.083 s
Version	z3-4.4.1

Figure 3: Output of signum.tl

- **Signum Function (buggy):** The code **signum_buggy.tl** also performs the signum function, but it has a bug in it, where it assigns sign -1 for signum(0). The constraint file remains the same as for signum.tl but this time our constraint should be satisfiable and z3 should also give us a set of values of variables where it fails. Figure 4 is the screenshot of the run of its code on Z3 Online Demostrator:



Input

SMT-LIB 2 script

```
(declare-fun x () Int)
(declare-fun sign1 () Int)
(declare-fun sign () Int)
(assert (= sign (ite (> x 0) 1 (- 1))))
(assert (= sign1 sign))
(assert (and (or (< x 0) (or (> x 0) (= x 0))) (and (or (< x 0) (or (> x 0) (not (=
sign1 0)))) (and (or (< x 0) (or (not (= sign1 1)) (= x 0))) (and (or (< x 0) (or
(not (= sign1 1)) (not (= sign1 0)))) (and (or (not (= sign1 (- 1))) (or (> x 0) (= x
0))) (and (or (not (= sign1 (- 1))) (or (> x 0) (not (= sign1 0)))) (and (or (not (=
sign1 (- 1))) (or (not (= sign1 1)) (= x 0))) (or (not (= sign1 (- 1))) (or (not (=
sign1 1)) (not (= sign1 0))))))))))))
(check-sat)
(get-model)
```

Output

Z3 output

```
sat
(model
  (define-fun x () Int
    0)
  (define-fun sign1 () Int
    (- 1))
  (define-fun sign () Int
    (- 1))
)
```

Summary

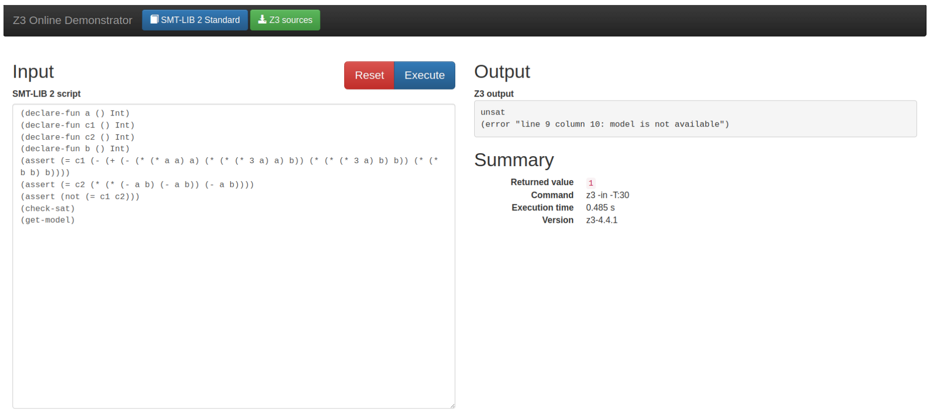
Command	z3 -in -T:30
Execution time	0.084 s
Version	z3-4.4.1

Figure 4: Output of signum_buggy.tl

- **Cube Identity:** The code **cube.tl** calculates $(a+b)^3$ using two ways. First by multiplying with itself 3 times, i.e., $(a+b)*(a+b)*(a+b)$. Second by using the algebraic identity

$$(a+b)^3 = a^3 - 3a^2b + 3ab - b^3$$

The constraint file simply checks if the two outputs are not equal. If we get such values of a and b, then the identity is incorrect. In this case, this does not happen as we can see in the figure Figure 5.



Input

SMT-LIB 2 script

```
(declare-fun a () Int)
(declare-fun c1 () Int)
(declare-fun c2 () Int)
(declare-fun b () Int)
(assert (= c1 (- (+ (- (* (* a a) a) (* (* (* 3 a) a) b)) (* (* (* 3 a) b) b)) (* (*
b b) b))))
(assert (= c2 (* (* (- a b) (- a b)) (- a b))))
(assert (not (= c1 c2)))
(check-sat)
(get-model)
```

Output

Z3 output

```
unsat
(error "line 9 column 10: model is not available")
```

Summary

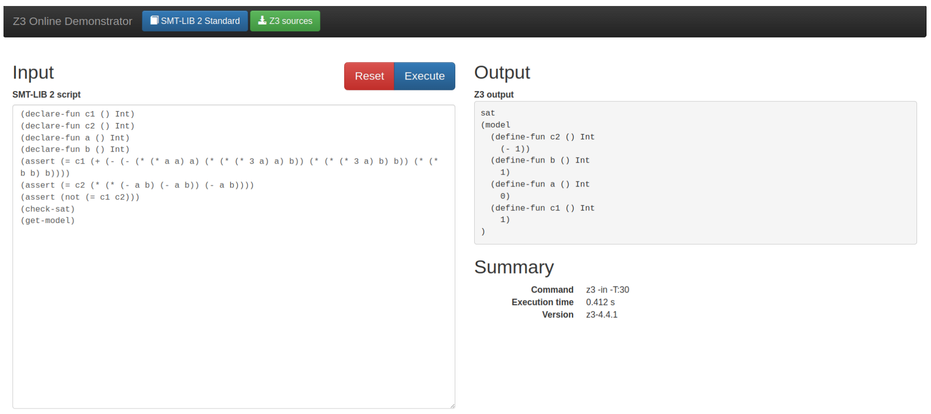
Returned value	1
Command	z3 -in -T:30
Execution time	0.485 s
Version	z3-4.4.1

Figure 5: Output of cube.tl

- **Cube Identity (Buggy):** The code **cube_buggy.tl** as in **cube.tl** except that we used a wrong identity

$$(a+b)^3 = a^3 - 3a^2b - 3ab + b^3 \quad (INCORRECT)$$

The constraint file is the same. In this case, we do find such values of a and b where identity fails as we can see in the figure Figure 6.



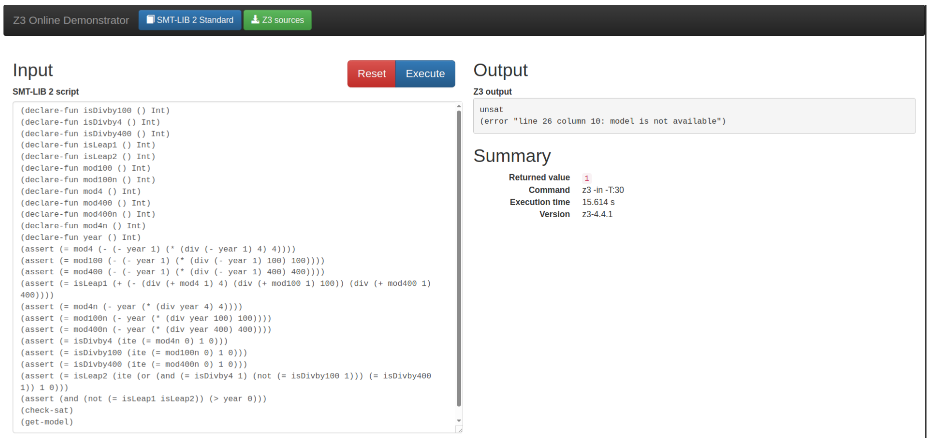
```
(declare-fun c1 () Int)
(declare-fun c2 () Int)
(declare-fun a () Int)
(declare-fun b () Int)
(assert (= c1 (+ (- (- (* (* a a) a) (* (* (* 3 a) a) b)) (* (* (* 3 a) b) b)) (* (*
b b) b))))
(assert (= c2 (* (* (- a b) (- a b)) (- a b))))
(assert (not (= c1 c2)))
(check-sat)
(get-model)
```

```
sat
(model
  (define-fun c2 () Int
    (- 1))
  (define-fun b () Int
    1)
  (define-fun a () Int
    0)
  (define-fun c1 () Int
    1)
)
```

```
      Command   z3 -in -T:30
Execution time  0.412 s
       Version  z3-4.4.1
```

Figure 6: Output of cube_buggy.tl

- **Leap Year Checker:** The code **leapyear.tl** compares two methods of checking whether a year is a leap year. The second method is a more intuitive way, written using only if-else statements and its correctness is ensured by the definition of leap year. The first method is rather non-intuitive and it checks leap year only using arithmetic assignment statements. In the constraint file, we check the negation of the condition: if the two outputs are the same (i.e. if result1 $\neq$ result2). If it is satisfiable, then our first method must be wrong and z3 would give us some value of year when first method would give wrong check of leap year. Figure 7 is the screenshot of the run of its code on Z3 Online Demostrator:



```
(declare-fun isDivby100 () Int)
(declare-fun isDivby4 () Int)
(declare-fun isDivby400 () Int)
(declare-fun isLeap1 () Int)
(declare-fun isLeap2 () Int)
(declare-fun mod100 () Int)
(declare-fun mod100n () Int)
(declare-fun mod4 () Int)
(declare-fun mod400 () Int)
(declare-fun mod400n () Int)
(declare-fun mod4n () Int)
(declare-fun year () Int)
(assert (= mod4 (- (- year 1) (* (div (- year 1) 4) 4))))
(assert (= mod100 (- (- year 1) (* (div (- year 1) 100) 100))))
(assert (= mod400 (- (- year 1) (* (div (- year 1) 400) 400))))
(assert (= isLeap1 (+ (- (div (+ mod4 1) 4) (div (+ mod100 1) 100)) (div (+ mod400 1)
400))))
(assert (= mod4n (- year (* (div year 4) 4))))
(assert (= mod100n (- year (* (div year 100) 100))))
(assert (= mod400n (- year (* (div year 400) 400))))
(assert (= isDivby4 (ite (= mod4n 0) 1 0)))
(assert (= isDivby100 (ite (= mod100n 0) 1 0)))
(assert (= isDivby400 (ite (= mod400n 0) 1 0)))
(assert (= isLeap2 (ite (or (and (= isDivby4 1) (not (= isDivby100 1))) (= isDivby400
1)) 1 0)))
(assert (and (not (= isLeap1 isLeap2)) (> year 0)))
(check-sat)
(get-model)
```

```
unsat
(error "line 26 column 10: model is not available")
```

```
Returned value  1
       Command  z3 -in -T:30
Execution time  15.614 s
       Version  z3-4.4.1
```

Figure 7: Output of leapyear.tl

How to run

Open the terminal in the home directory of the git repo. Run the following commands:

```
cd ChironCore
python3 chiron.py -smt <path-to-constraint-file> <path-to-output-file> <path
    -to-src-program-file>
```

For initialization of variables, use the `-d/--params` flag:

```
cd ChironCore
python3 chiron.py -smt <path-to-constraint-file> <path-to-output-file> -d <
    python-dictionary-of-params> <path-to-src-program-file>
```

Further Tasks

- **Loops:** We will generate SMTLIB code for loop (repeat statement in Chiron). We would also need a pre-condition (to be satisfied before entering the loop), a post-condition (to be satisfied after exiting the loop) and the Invariant to be satisfied on each iteration of the loop. We will modify the format of constraint file and generate SMTLIB code for this to support looping in the code.
- **Turtle Commands:** We will generate SMTLIB code for turtle commands : `penup, pendown, forward, backward, left, right, goto`.
- **Integration of Z3 solver:** We will integrate python Z3 solver into the project.
- **Nested if-else blocks:** We will lift some of the assumptions and support nested if-else blocks.

References

- AST Visitor Template: https://gist.github.com/jtpio/cb30bca7abeceae0234c9ef43eec28b4
- AST Documentation: https://docs.python.org/3/library/ast.htmlast.NodeVisitor.visit
- SMTLIB Documentation: https://smt-lib.org/
- NetworkX Documentation: https://networkx.org/documentation/stable/
- Z3 Online Demonstration: https://compsys-tools.ens-lyon.fr/z3/
- Z3 Playground: https://jfmc.github.io/z3-play/
- Python Regular Expressions: https://docs.python.org/3/library/re.html
- Python Official Website: https://www.python.org/