Problem Solving with SMT Solvers

Ming-Hsien Tsai, FLOLAC 2025

Satisfiability Modulo Theories

- *Satisfiability modulo theories (SMT)* is the problem of determining whether a mathematical formula is satisfiable
  - The formula is interpreted within theories in first-order logic with equality
- SMT solvers implement decision procedures for various first-order theories
  - Alt-ergo, Boolector, CVC5, MathSAT, Yices, Z3
- Many SMT solvers implement a common interface format called *SMTLIB2* (https://smt-lib.org/)
- Some SMT solvers may additionally support its own interface formats
  - Btor and Btor2 for Boolector

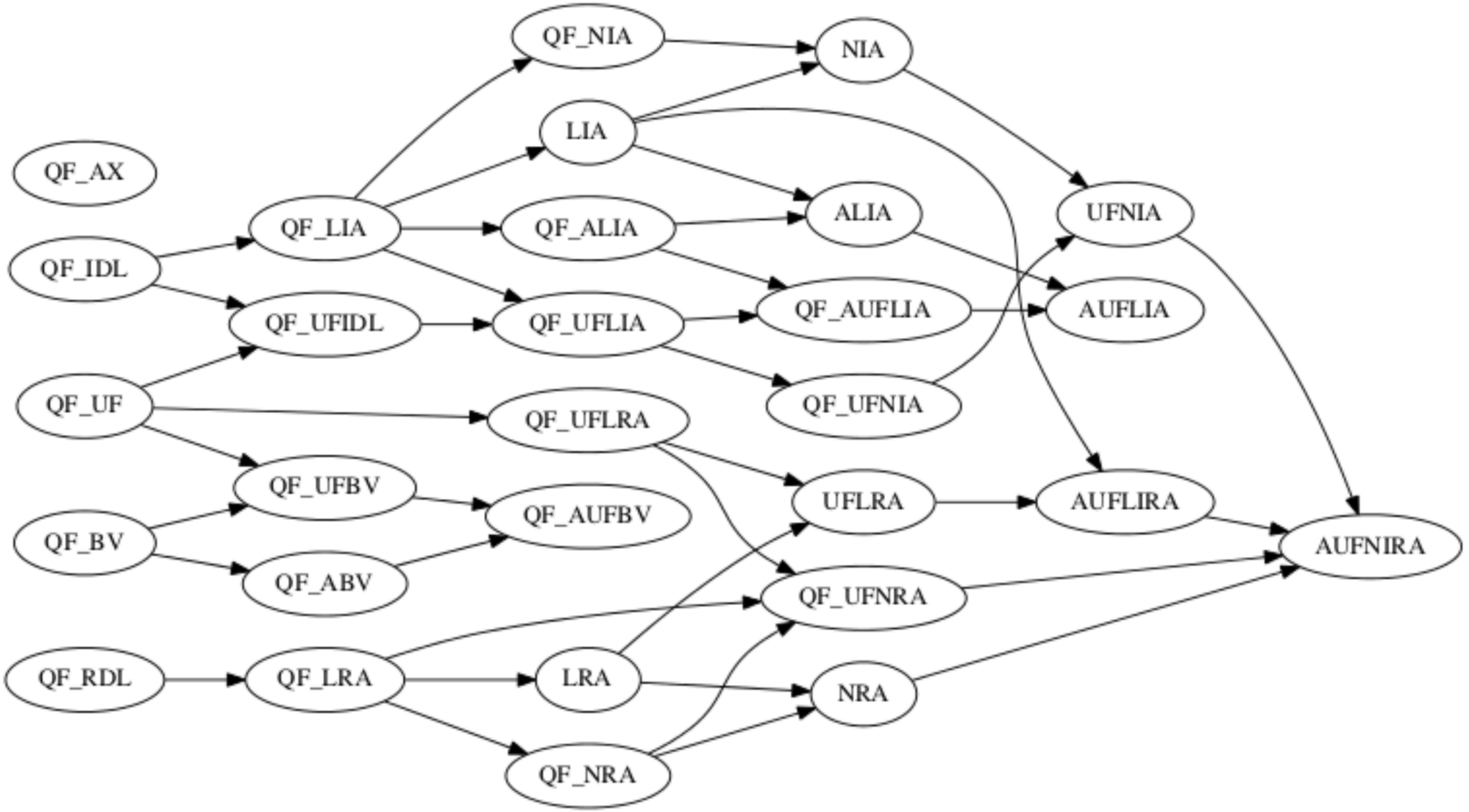
SMT-LIB Theories

- **ArraysEx**
  - Functional arrays with extensionality
- **FixedSizeBitVectors**
  - Bit vectors with arbitrary size
- **Core**
  - Core theory, defining the basic Boolean operators
- **FloatingPoint**
  - Floating point numbers

SMT-LIB Theories (cont'd)

- **Ints**
  - Integer numbers
- **Reals**
  - Real numbers
- **Reals_Ints**
  - Real and integer numbers
- **Strings**
  - Unicode character strings and regular expressions

SMT-LIB Logics



Naming Conventions

- **QF** for the restriction to quantifier free formulas
- **A** or **AX** for the theory ArraysEx
- **BV** for the theory FixedSizeBitVectors
- **FP** (forthcoming) for the theory FloatingPoint
- **IA** for the theory Ints (Integer Arithmetic)
- **RA** for the theory Reals (Real Arithmetic)

Naming Conventions (cont'd)

- **IRA** for the theory Reals_Ints (mixed Integer Real Arithmetic)
- **IDL** for Integer Difference Logic
- **RDL** for Rational Difference Logic
- **L** before IA, RA, or IRA for the linear fragment of those arithmetics
- **N** before IA, RA, or IRA for the non-linear fragment of those arithmetics
- **UF** for the extension allowing free sort and function symbols

SMT-LIB Format

- The syntax of the SMT-LIB language is similar to that of the LISP programming language
  - A sequence of one or more S-expressions
  - An S-expression is either
    - a token, or
    - a sequence of 0 or more S-expressions enclosed in a pair of left and right parentheses
  - Line comments start with ";"
- Examples:
  - `(abc x y z)`
  - `(+ 5 (* 2 3))`

An SMT-LIB Example

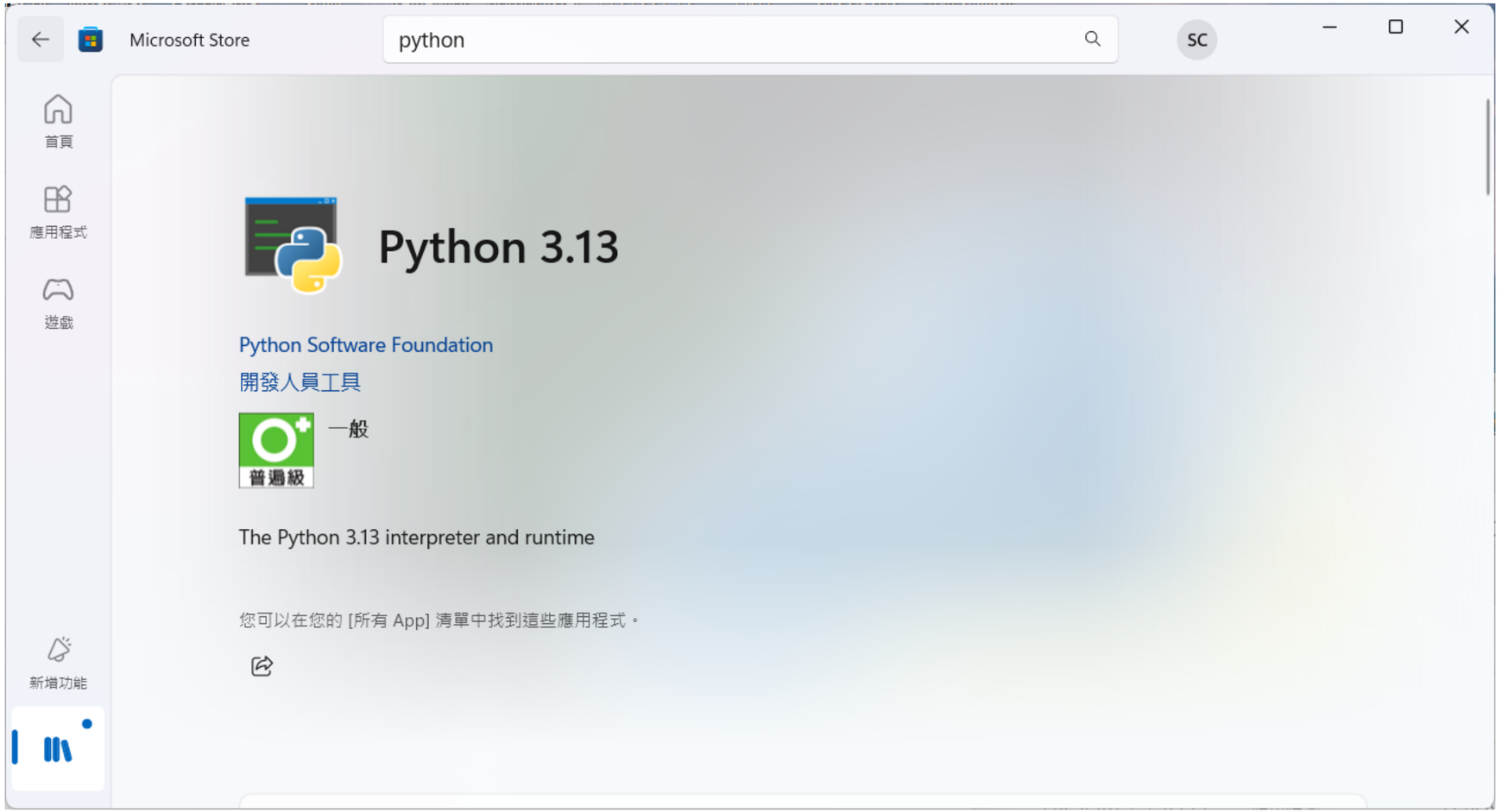
```
(set-logic LIA)                          ; Explicitly set the logic
(set-option :produce-models true) ; Generate satisfying assignments
(declare-fun x () Int)                   ; Declare a 0-ary function symbol x
(declare-const y Int)                    ; Declare a constant y
(assert (< (+ x y) 10))                  ; Assert x + y < 10
(assert (> x 4))                         ; Assert x > 4
(check-sat)                              ; Check satisfiability
(get-model)                              ; Print satisfying assignment
```

Z3

- Z3 is a high performance theorem prover developed at Microsoft Research
- Z3 provides API interfaces for several programming languages
  - Python (Z3Py)
  - C
  - .Net
  - OCaml
  - Scalar

Installation of Z3+Z3py on Windows

- Step 1: Install Python from Windows Store -

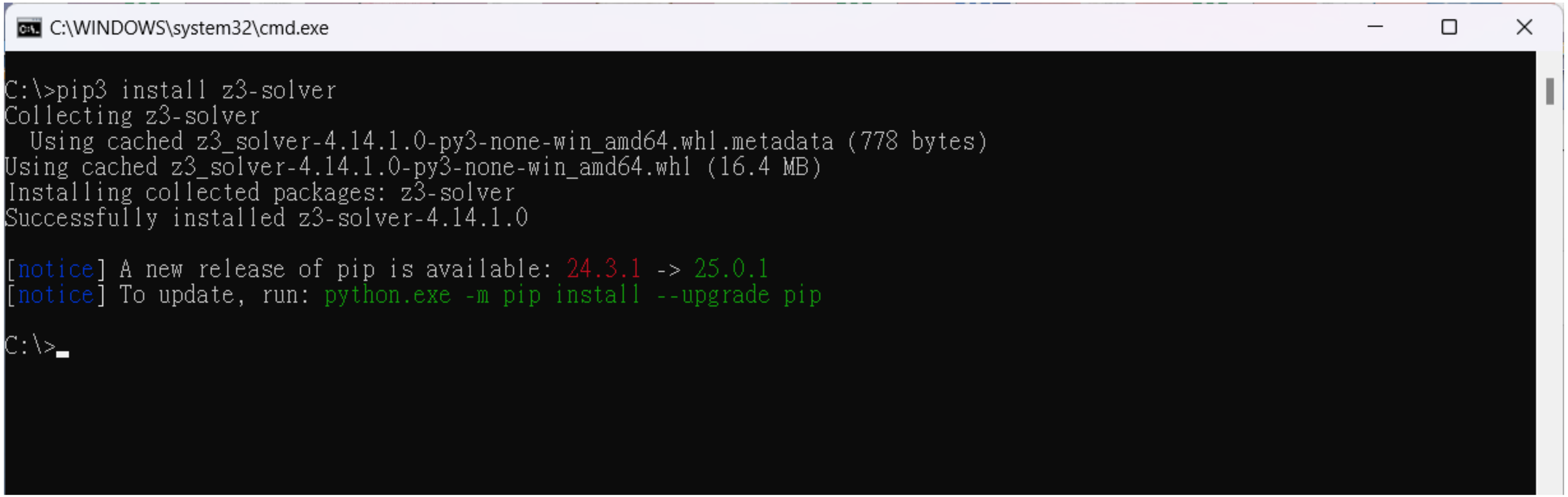


Installation of Z3+Z3py on Windows

- Step 1 (alternative): Install Python from python.org -

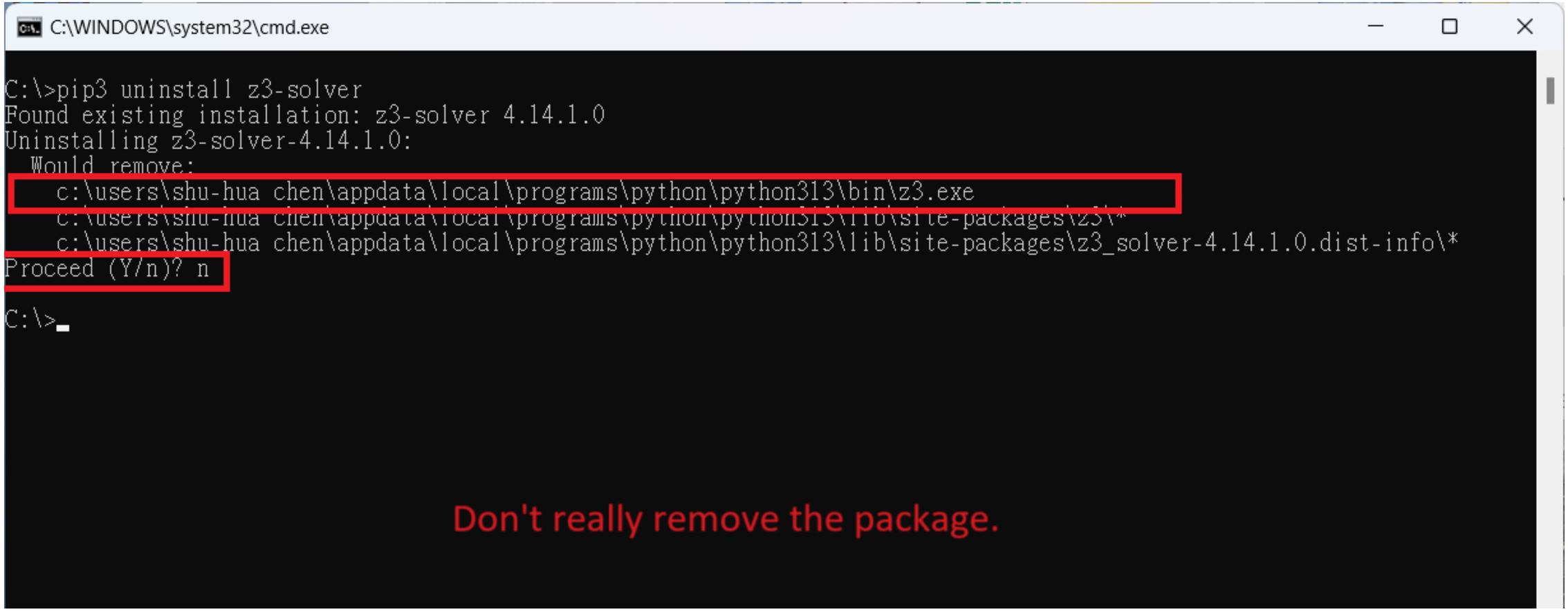
Installation of Z3+Z3py on Windows

- Step 2: Install z3-solver using pip3 -



Installation of Z3+Z3py on Windows

- Step 3: Find where z3.exe is installed -



Installation of Z3+Z3py on Windows

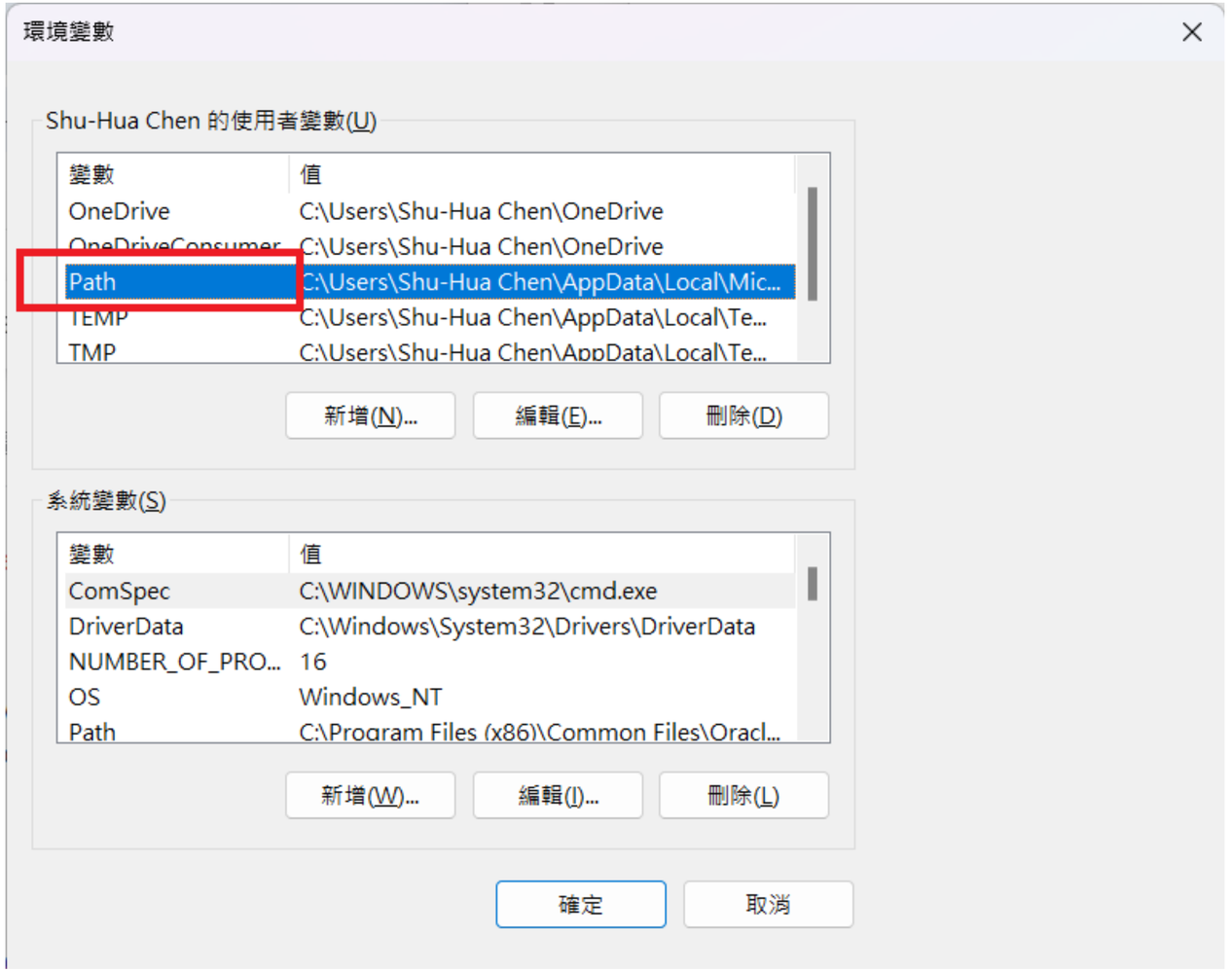
- Step 4: Open Settings > System > About > Advanced system settings -

Installation of Z3+Z3py on Windows

- Step 5: Open Environment Variables -

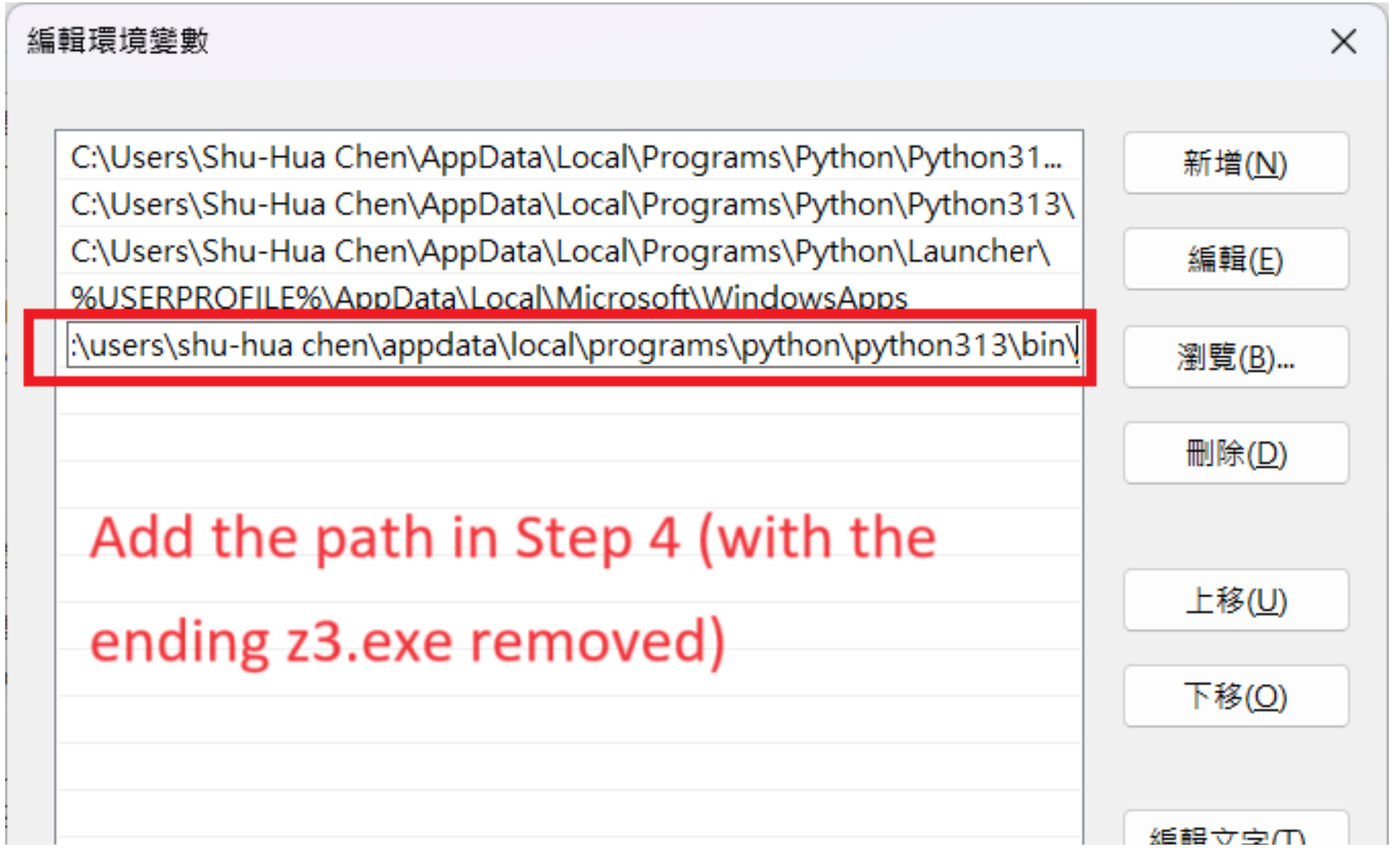
Installation of Z3+Z3py on Windows

- Step 6: Edit the environment variable Path -



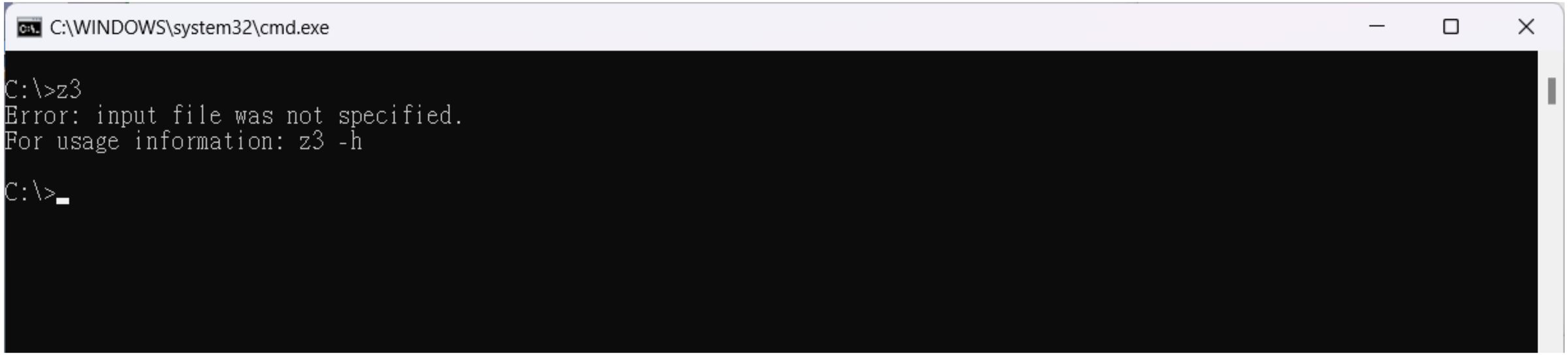
Installation of Z3+Z3py on Windows

- Step 7: Add the path to z3.exe -



Installation of Z3+Z3py on Windows

- Step 8: Close and reopen terminal -



Installation of Z3+Z3py on Windows-WSL/Linux/macOS

- Run the following console command
  - pip3 install z3-solver
- You may use a virtual environment (https://docs.python.org/3/library/venv.html) for Python
- Don't forget to add the Python bin path to the environment variable PATH

```
🐬 mht208@IrukaMBP ~ 🐬
 $ python3 -c "import sys
print(sys.prefix)"
/opt/homebrew/opt/python@3.13/Frameworks/Python.fr
amework/Versions/3.13

🐬 mht208@IrukaMBP ~ 🐬
 $ ls /opt/homebrew/opt/python@3.13/Frameworks/Pyt
hon.framework/Versions/3.13/bin
idle3             pydoc3.13
idle3.13          python3
```

```
🐬 mht208@IrukaMBP ~ 🐬
 $ python3 -c "import sys
dquote> print(sys.prefix)"

🐬 mht208@IrukaMBP ~ 🐬
 $ ls /Users/mht208/.local/opt/python-venv/bin
Activate.ps1          nc3tonc4
__pycache__           nc4tonc3
activate              ncinfo
```

First Example

- `from z3 import *`: import z3 API
- `x = Int('x')`: create an integer variable named x
- `solve(...)`: solve a system of constraints

```
from z3 import *

x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

Code

```
$ python3 example-01-int-solve.py
[y = 0, x = 7]
```

Output

Formula/Expression Simplifier

```
x = Int('x')
y = Int('y')
print (simplify(x + y + 2*x + 3))
print (simplify(x < y + x + 2))
print (simplify(And(x + 1 >= 3, x**2 + x**2 + y**2 + 2 >= 5)))
```

Code

```
 $ python3 example-02-simplify.py
3 + 3*x + y
Not(y <= -2)
And(x >= 2, 2*x**2 + y**2 >= 3)
```

Output

Display Mode

```
x = Int('x')
y = Int('y')
set_option(html_mode=True)
print (And(x**2 + y**2 >= 1, x > 2))
set_option(html_mode=False)
print (And(x**2 + y**2 >= 1, x > 2))
```

Code

```
$ python3 example-03-html-mode.py
x<sup>2</sup> + y<sup>2</sup> &ge; 1 &and; x &gt; 2
And(x**2 + y**2 >= 1, x > 2)
```

Output

Expression Traversal

```
x = Int('x')
y = Int('y')
n = x + y >= 3
print ("num args: ", n.num_args())
print ("children: ", n.children())
print ("1st child:", n.arg(0))
print ("2nd child:", n.arg(1))
print ("operator: ", n.decl())
print ("op name:  ", n.decl().name())
```

Code

```
$ python3 example-04-expression-traversal.py
num args:  2
children:  [x + y, 3]
1st child: x + y
2nd child: 3
operator:  >=
op name:   >=
```

Output

Rational Numbers

```
x = Real('x')
y = Real('y')
solve(x**2 + y**2 > 3, x**3 + y < 5)
```

Code

```
$ python3 example-05-rational.py
[y = 2, x = 1/8]
```

Output

Z3Py can represent arbitrarily large integers, rational numbers, and irrational algebraic numbers

Irrational Numbers

```
x = Real('x')
y = Real('y')
solve(x**2 + y**2 == 3, x**3 == 2)

set_option(precision=30)
print ("Solving, and displaying result with 30 decimal places")
solve(x**2 + y**2 == 3, x**3 == 2)
```

Code

```
 $ python3 example-06-irrational.py
[y = -1.1885280594?, x = 1.2599210498?]
Solving, and displaying result with 30 decimal places
[y = -1.188528059421316533710369365015?,
 x = 1.259921049894873164767210607278?]
```

Output

Create Rational Numbers

```
RealVal(n) / m
n / RealVal(m)
Q(n, m)
```

```
print (1/3)
print (RealVal(1)/3)
print (Q(1,3))

x = Real('x')
print (x + 1/3)
print (x + Q(1,3))
print (x + "1/3")
print (x + 0.25)
```

Code

```
$ python3 example-07-create-real-numbers.py
0.3333333333333333
1/3
1/3
x + 3333333333333333/10000000000000000
x + 1/3
x + 1/3
x + 1/4
```

Output

Rational Numbers in Decimal Notation

```
x = Real('x')
solve(3*x == 1)

set_option(rational_to_decimal=True)
solve(3*x == 1)

set_option(precision=30)
solve(3*x == 1)
```

Code

```
$ python3 example-08-rational-decimal-notation.py
[x = 1/3]
[x = 0.3333333333?]
[x = 0.333333333333333333333333333333?]
```

Output

Boolean Logic

- Z3 supports Boolean operators: `And`, `Or`, `Not`, `Implies`, `If`
- Bi-implications are represented using equality `==`
- The Python Boolean constants `True` and `False` can be used to build Z3 Boolean expressions

```
p = Bool('p')
q = Bool('q')
r = Bool('r')
solve(Implies(p, q), r == Not(q), Or(Not(p), r))

print (And(p, q, True))
print (simplify(And(p, q, True)))
print (simplify(And(p, False)))
```

Combination of Polynomial and Boolean Constraints

```
p = Bool('p')
x = Real('x')
solve(Or(x < 5, x > 10), Or(p, x**2 == 2), Not(p))
```

Solvers

- The command `Solver()` creates a general purpose solver
- Constraints can be added using the method `add`
- The result is `sat` (satisfiable), `unsat` (unsatisfiable), or `unknown`
- The command `push` creates a new scope by saving the current stack size
- The command `pop` removes any assertion performed between it and the matching push
- The `check` method always operates on the content of solver assertion stack

Basic Solver API

```
x = Int('x')
y = Int('y')

s = Solver()
print (s)

s.add(x > 10, y == x + 2)
print (s)
print ("Solving constraints in the solver s ...")
print (s.check())

print ("Create a new scope...")
s.push()
s.add(y < 11)
print (s)
print ("Solving updated set of constraints...")
print (s.check())

print ("Restoring state...")
s.pop()
print (s)
print ("Solving restored set of constraints...")
print (s.check())
```

Code

```
 $ python3 example-11-basic-solver-api.py
[]
[x > 10, y == x + 2]
Solving constraints in the solver s ...
sat
Create a new scope...
[x > 10, y == x + 2, y < 11]
Solving updated set of constraints...
unsat
Restoring state...
[x > 10, y == x + 2]
Solving restored set of constraints...
sat
```

Output

Unknown Result

```
x = Real('x')
s = Solver()
s.add(2**x == 3)
print (s.check())
```

Code

```
$ python3 example-12-unknown.py
unknown
```

Output

Performance Statistics

```
x = Real('x')
y = Real('y')
s = Solver()
s.add(x > 1, y > 1, Or(x + y > 3, x - y < 2))
print ("asserted constraints...")
for c in s.assertions():
    print (c)

print (s.check())
print ("statistics for the last check method...")
print (s.statistics())
# Traversing statistics
for k, v in s.statistics():
    print ("%s : %s" % (k, v))
```

Code

```
$ python3 example-13-statistics.py
asserted constraints...
x > 1
y > 1
Or(x + y > 3, x - y < 2)
sat
statistics for the last check method...
(:arith-lower          1
 :arith-make-feasible 3
 :arith-max-columns   8
 :arith-max-rows      2
 :arith-upper         3
 :decisions           2
 :final-checks        1
 :max-memory          21.31
 :memory              20.31
 :mk-bool-var         4
```

Output

Model

```
x, y, z = Reals('x y z')
s = Solver()
s.add(x > 1, y > 1, x + y > 3, z - x < 10)
print (s.check())

m = s.model()
print ("x = %s" % m[x])

print ("traversing model...")
for d in m.decls():
    print ("%s = %s" % (d.name(), m[d]))
```

Code

```
 $ python3 example-14-model.py
sat
x = 3/2
traversing model...
y = 2
x = 3/2
z = 0
```

Output

Combine Real and Integer Numbers

`ToReal(e)`: cast an integer expression into a real expression

```
x = Real('x')
y = Int('y')
a, b, c = Reals('a b c')
s, r = Ints('s r')
print (x + y + 1 + (a + s))
print (ToReal(y) + c)
```

Code

```
$ python3 example-15-real-integer.py
x + ToReal(y) + 1 + a + ToReal(s)
ToReal(y) + c
```

Output

Simplify with Transformation

Code

```
x, y = Reals('x y')
# Put expression in sum-of-monomials form
t = simplify((x + y)**3, som=True)
print (t)
# Use power operator
t = simplify(t, mul_to_power=True)
print (t)
```

Output

```
 $ python3 example-16-simplify-transformation.py
x*x*x + 3*x*x*y + 3*x*y*y + y*y*y
x**3 + 3*x**2*y + 3*x*y**2 + y**3
```

Options for Simplify

Code

```
x, y = Reals('x y')
# Using Z3 native option names
print (simplify(x == y + 2, ':arith-lhs', True))
# Using Z3Py option names
print (simplify(x == y + 2, arith_lhs=True))

print ("\nAll available options:")
help_simplify()
```

Output

```
 $ python3 example-17-options-simplify.py
x + -1*y == 2
x + -1*y == 2

All available options:
algebraic_number_evaluator (bool) simplify/evaluate expressions containing (algebrai
c) irrational numbers. (default: true)
arith_ineq_lhs (bool) rewrite inequalities so that right-hand-side is a constant. (d
efault: false)
arith_lhs (bool) all monomials are moved to the left-hand-side, and the right-hand-s
ide is just a constant. (default: false)
```

Options for Simplify

Code

```
x, y = Reals('x y')
solve(x + 10000000000000000000000 == y, y > 2000000000000000)

print (Sqrt(2) + Sqrt(3))
print (simplify(Sqrt(2) + Sqrt(3)))
print (simplify(Sqrt(2) + Sqrt(3)).sexpr())
# The sexpr() method is available for any Z3 expression
print ((x + Sqrt(y) * 2).sexpr())
```

Output

```
 $ python3 example-18-large-numbers.py
[y = 2000000000000001, x = -999997999999999999999]
2**(1/2) + 3**(1/2)
3.1462643699?
(root-obj (+ (^ x 4) (* (- 10) (^ x 2)) 1) 4)
(+ x (* (^ y (/ 1.0 2.0)) 2.0))
```

Z3Py only supports algebraic irrational numbers

Machine Arithmetic

- Machine arithmetic is available in Z3Py as Bit-Vectors
- They implement the precise semantics of unsigned and of signed two-complements arithmetic

```
 $ python3 example-19-bitvec.py
x + 2
(bvadd x #x0002)
65535 + x + y
True
False
```

Output

```
x = BitVec('x', 16)
y = BitVec('y', 16)
print (x + 2)
# Internal representation
print ((x + 2).sexpr())

# -1 is equal to 65535 for 16-bit integers
print (simplify(x + y - 1))

# Creating bit-vector constants
a = BitVecVal(-1, 16)
b = BitVecVal(65535, 16)
print (simplify(a == b))

a = BitVecVal(-1, 32)
b = BitVecVal(65535, 32)
# -1 is not equal to 65535 for 32-bit integers
print (simplify(a == b))
```

Code

Machine Arithmetic

- In Z3Py, the operators <, <=, >, >=, /, % and >> correspond to the signed versions
- The corresponding unsigned operators are ULT, ULE, UGT, UGE, UDiv, URem and LShR.

4294967295 = #xFFFFFFFF

```
$ python3 example-20-bitvec-compare.py
[y = 1, x = 1]
[x = 0, y = 4294967295]
[x = 4294967295]
no solution
```

Output

```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)

solve(x + y == 2, x > 0, y > 0)

# Bit-wise operators
# & bit-wise and
# | bit-wise or
# ~ bit-wise not
solve(x & y == ~y)

solve(x < 0)

# using unsigned version of <
solve(ULT(x, 0))
```

Code

Machine Arithmetic (cont'd)

Code

```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)
solve(x >> 2 == 3)
solve(x << 2 == 3)
solve(x << 2 == 24)
solve(x >> 2 < 0)
solve(LShR(x, 2) < 0)
```

Output

```
 $ python3 example-21-bitvec-shift.py
[x = 12]
no solution
[x = 6]
[x = 2147483648]
no solution
```

Functions

Code

```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
solve(f(f(x)) == x, f(x) == y, x != y)
```

Output

```
$ python3 example-22-functions.py
[x = 0, y = 1, f = [1 -> 0, else -> 1]]
```

Expression Evaluation in Model

Code

```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
s = Solver()
s.add(f(f(x)) == x, f(x) == y, x != y)
print (s.check())
m = s.model()
print ("f(f(x)) =", m.evaluate(f(f(x))))
print ("f(x)    =", m.evaluate(f(x)))
```

Output

```
 $ python3 example-23-eval-expr.py
sat
f(f(x)) = 0
f(x)    = 1
```

Satisfiability and Validity

- A formula/constraint F is valid if F always evaluates to true for any assignment of appropriate values to its uninterpreted symbols
- A formula/constraint F is satisfiable if there is some assignment of appropriate values to its uninterpreted symbols under which F evaluates to true
- Validity is about finding a proof of a statement; satisfiability is about finding a solution to a set of constraints
- F is valid precisely when `Not(F)` is not satisfiable

Prove Validity

Code

```
p, q = Bools('p q')
demorgan = And(p, q) == Not(Or(Not(p), Not(q)))
print (demorgan)

print("Check satisfiability of the negation...")
solve(Not(demorgan))

print ("Proving demorgan...")
prove(demorgan)
```

Output

```
 $ python3 example-24-prove.py
And(p, q) == Not(Or(Not(p), Not(q)))
Check satisfiability of the negation...
no solution
Proving demorgan...
proved
```

List Comprehensions

```
# Create list [1, ..., 5]
print ([ x + 1 for x in range(5) ])

# Create two lists containing 5 integer variables
X = [ Int('x%s' % i) for i in range(5) ]
Y = [ Int('y%s' % i) for i in range(5) ]
print (X)

# Create a list containing X[i]+Y[i]
X_plus_Y = [ X[i] + Y[i] for i in range(5) ]
print (X_plus_Y)

# Create a list containing X[i] > Y[i]
X_gt_Y = [ X[i] > Y[i] for i in range(5) ]
print (X_gt_Y)

print (And(X_gt_Y))

# Create a 3x3 "matrix" (list of lists) of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(3) ]
      for i in range(3) ]
pp(X)
```

Code

`pp` is similar to `print`, but it uses Z3Py formatter for lists and tuples

```
$ python3 example-25-list-comprehensions.py
[1, 2, 3, 4, 5]
[x0, x1, x2, x3, x4]
[x0 + y0, x1 + y1, x2 + y2, x3 + y3, x4 + y4]
[x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4]
And(x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4)
[[x_1_1, x_1_2, x_1_3],
 [x_2_1, x_2_2, x_2_3],
 [x_3_1, x_3_2, x_3_3]]
```

Output

Vectors

Create a list of variables without worrying about the variable names

Code

```
X = IntVector('x', 5)
Y = RealVector('y', 5)
P = BoolVector('p', 5)
print (X)
print (Y)
print (P)
print ([ y**2 for y in Y ])
print (Sum([ y**2 for y in Y ]))
```

Output

```
 $ python3 example-26-vectors.py
[x__0, x__1, x__2, x__3, x__4]
[y__0, y__1, y__2, y__3, y__4]
[p__0, p__1, p__2, p__3, p__4]
[y__0**2, y__1**2, y__2**2, y__3**2, y__4**2]
y__0**2 + y__1**2 + y__2**2 + y__3**2 + y__4**2
```

Power of Two

- A trick in testing whether a machine integer is a power of two

Code

```
x      = BitVec('x', 32)
powers = [ 2**i for i in range(32) ]
fast   = And(x != 0, x & (x - 1) == 0)
slow   = Or([ x == p for p in powers ])
print (fast)
prove(fast == slow)

print ("trying to prove buggy version...")
fast   = x & (x - 1) == 0
prove(fast == slow)
```

Output

```
 $ python3 example-27-power-of-two.py
And(x != 0, x & x - 1 == 0)
proved
trying to prove buggy version...
counterexample
[x = 0]
```

Puzzle: Dog, Cat and Mouse

- Spend exactly 100 dollars and buy exactly 100 animals. Dogs cost 15 dollars, cats cost 1 dollar, and mice cost 25 cents each. You have to buy at least one of each. How many of each should you buy?

Application: Install Problem

- Determine whether a new set of packages can be installed in a system
  - The depends clauses stipulate which other packages must be present
  - The conflicts clauses stipulate which other packages must not be present

Distribution Rules	Constraints
`Package: a`	
`Depends: b,`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{b}})$
`c,`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{c}})$
`z`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{z}})$
`Package: b`	
`Depends: d`	$(\neg x_{\mathtt{b}} \vee x_{\mathtt{d}})$
`Package: c`	
`Depends: d | e,`	$(\neg x_{\mathtt{c}} \vee x_{\mathtt{d}} \vee x_{\mathtt{e}})$
`f | g`	$(\neg x_{\mathtt{c}} \vee x_{\mathtt{f}} \vee x_{\mathtt{g}})$
`Package: d`	
`Conflicts: e`	$(\neg x_{\mathtt{d}} \vee \neg x_{\mathtt{e}})$

Can we install a, z, and g together?

Tactics

- Tactics: "big" symbolic reasoning steps represented as functions
- Tactics process sets of formulas called Goals
- When a tactic is applied to some goal G, four different outcomes are possible
  - Succeed in showing G to be satisfiable
  - Succeed in showing G to be unsatisfiable
  - Produce a sequence of subgoals
  - Fail
- The command `describe_tactics()` provides a short description of all built-in tactics

Tactic Combinators

- Z3Py comes equipped with the following tactic combinators (aka tacticals):
  - `Then(t, s)`: Apply t to the input goal and s to every subgoal produced by t
  - `OrElse(t, s)`: First apply t to the given goal, if it fails then returns the result of s applied to the given goal
  - `Repeat(t)`: Keep applying the given tactic until no subgoal is modified by it
  - `Repeat(t, n)`: Keep applying the given tactic until no subgoal is modified by it, or the number of iterations is greater than n
  - `TryFor(t, ms)`: Apply tactic t to the input goal, if it does not return in ms millisenconds, it fails
  - `With(t, params)`: Apply the given tactic using the given parameters

Tactics: solve-eqs

- The tactic `solve-eqs` eliminate variables using Gaussian elimination

Code

```
x, y = Reals('x y')
g  = Goal()
g.add(x > 0, y > 0, x == y + 2)
print (g)

t1 = Tactic('simplify')
t2 = Tactic('solve-eqs')
t  = Then(t1, t2)
print (t(g))
```

Output

```
 $ python3 example-29-tactics-solve-eqs.py
[x > 0, y > 0, x == y + 2]
[[Not(x <= 0), Not(x <= 2)]]
```

Tactics: split-clause

- The tactic `split-clause` will select a clause `Or(f_1, ..., f_n)` in the input goal, and split it n subgoals

Code

```
x, y = Reals('x y')
g  = Goal()
g.add(Or(x < 0, x > 0), x == y + 1, y < 0)

t = Tactic('split-clause')
r = t(g)
for g in r:
    print (g)
```

Output

```
 $ python3 example-30-tactics-split-clause.py
[x < 0, x == y + 1, y < 0]
[x > 0, x == y + 1, y < 0]
```

Combine Tactics and Solver

```
t = Then('simplify',
         'normalize-bounds',
         'solve-eqs')

x, y, z = Ints('x y z')
g = Goal()
g.add(x > 10, y == x + 3, z > y)

r = t(g)
# r contains only one subgoal
print (r)

s = Solver()
s.add(r[0])
print (s.check())
# Model for the subgoal
print (s.model())
# Model for the original goal
print (r[0].convert_model(s.model()))
```

Code

`convert_model`: convert a model for a subgoal into a model for the original goal

```
$ python3 example-31-combine-tactics-solver.py
[[Not(y <= 13), Not(z <= y)]]
sat
[y = 14, z = 15]
[z = 15, y = 14, x = 11]
```

Output

Uninterpreted Sorts

```
A    = DeclareSort('A')
x, y = Consts('x y', A)
f    = Function('f', A, A)

s    = Solver()
s.add(f(f(x)) == x, f(x) == y, x != y)

print (s.check())
m = s.model()
print (m)
print ("interpretation assigned to A:")
print (m[A])
```

Code

```
 $ python3 example-32-uninterpreted-sorts.py
sat
[x = A!val!0,
 y = A!val!1,
 f = [A!val!1 -> A!val!0, else -> A!val!1]]
interpretation assigned to A:
[A!val!0, A!val!1]
```

Output

Quantifiers

Code

```
f = Function('f', IntSort(), IntSort(), IntSort())
x, y = Ints('x y')
print (ForAll([x, y], f(x, y) == 0))
print (Exists(x, f(x, x) >= 0))

a, b = Ints('a b')
solve(ForAll(x, f(x, x) == 0), f(a, b) == 1)
```

Output

```
 $ python3 example-33-quantifiers.py
ForAll([x, y], f(x, y) == 0)
Exists(x, f(x, x) >= 0)
[b = 2, a = 0, f = [(0, 2) -> 1, else -> 0]]
```

Datatypes: Color

```
# Declare Color
Color = Datatype('Color')
# Color constructors
Color.declare('red')
Color.declare('green')
Color.declare('blue')
# Create the datatype
Color = Color.create()

print (is_expr(Color.green))
print (Color.green == Color.blue)
print (simplify(Color.green == Color.blue))

# Let c be a constant of sort Color
c = Const('c', Color)
# Then, c must be red, green or blue
prove(Or(c == Color.green,
         c == Color.blue,
         c == Color.red))
```

Code

```
 $ python3 example-34-datatypes-color.py
True
green == blue
False
proved
```

Output

Datatypes: EnumSort

Code

```
Color, (red, green, blue) = EnumSort('Color', ('red', 'green', 'blue'))

print (green == blue)
print (simplify(green == blue))

c = Const('c', Color)
solve(c != green, c != blue)
```

Output

```
 $ python3 example-35-enum-sort.py
green == blue
False
[c = red]
```

Datatypes: Lists

```
def DeclareList(sort):
    List = Datatype('List_of_%s' % sort.name())
    List.declare('cons', ('car', sort), ('cdr', List))
    List.declare('nil')
    return List.create()

IntList     = DeclareList(IntSort())
RealList    = DeclareList(RealSort())
IntListList = DeclareList(IntList)

l1 = IntList.cons(10, IntList.nil)
print (l1)
print (simplify(IntList.car(l1)))
print (IntListList.cons(l1, IntListList.cons(l1, IntListList.nil)))
print (RealList.cons("1/3", RealList.nil))
print (l1.sort())
```

Code

```
$ python3 example-36-datatypes-list.py
cons(10, nil)
10
cons(cons(10, nil), cons(cons(10, nil), nil))
cons(1/3, nil)
List_of_Int
```

Output

Z3Py Exercise - Linear Integer Arithmetic

- Solve the following satisfiability problem by Z3Py
  - There are three distinct integers x, y, and z
  - $x + y + z = 10$
  - $0 \leq x \leq 10$
  - $0 \leq y \leq 10$
  - $0 \leq z \leq 10$

Z3Py Exercise - Integer Arithmetic

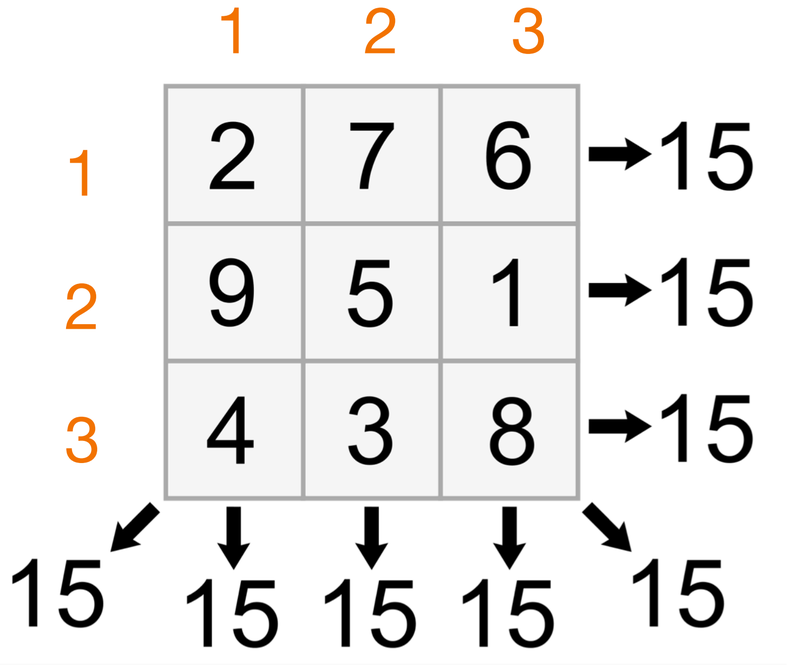
- Prove the following implications by Z3Py (all variables are integers)
  - $x + y + z = 0$
    $\rightarrow (x \cdot y) + (x \cdot z) + (y \cdot z) = 0$
    $\rightarrow x \cdot y \cdot z = 0$
    $\rightarrow x^3 = 0$
  - $x^2 + x \cdot y = 0$
    $\rightarrow y^2 + x \cdot y = 0$
    $\rightarrow x + y = 0$
- Hint: show the negation is unsatisfiable

Z3Py Exercise - Uninterpreted Function

- There is a function $f : A \to A$
- Prove that
- $$\begin{aligned} & f(f(x)) = x \\ \to\ & f(f(f(x))) = x \\ \to\ & f(x) = x \end{aligned}$$

Z3Py Exercise - 3x3 Magic Square

- Solve the 3x3 magic square problem using Z3Py
  - An example of 3x3 magic square



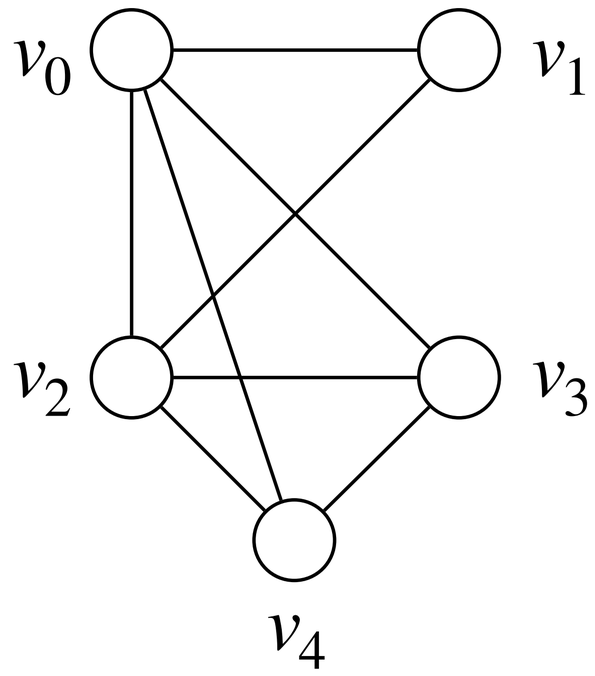
Z3Py Exercise - Sudoku

- Solve the Sudoku problem using Z3Py

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Z3Py Exercise - Graph Coloring Problem

- Solve the graph coloring problem using Z3Py
  - $G = (V, E)$ contains 5 vertices and 8 edges
  - $K = \{k_0, k_1, k_2\}$



Z3Py Exercise - Swap (Shallow Embedding)

- Assume swap is a function that swaps two integers in a pair
- Prove that for all pair p, swap(swap(p)) = p
- Shallow embedding:
  - Declare sort Pair
  - Declare functions to construct Pair and to access pair elements
  - Declare function swap
  - Assert properties of swap
  - Assert it is not the case that for all pair p, swap(swap(p)) = p

Z3Py Exercise - Swap (Deep Embedding)

- Assume swap is a function that swaps two integers in a pair
- Prove that for all pair p, swap(swap(p)) = p
- Deep embedding:
  - Declare datatype Pair
  - Define function swap
  - Assert it is not the case that for all pair p, swap(swap(p)) = p

Z3Py Exercise - Max

- Verify the assertion of the following program (assuming that x is an integer)

```
if (x > y)
  z = x;
else
  z = y;
assert(z >= x && z >= y);
```

Z3Py Exercise - Abs

- Verify the assertion of the following program (assuming that x is a 32-bit signed integer)

```
if (x < 0)
  y = - x;
else
  y = x;
assert(y >= 0);
```

Z3Py Exercise - ecp_nistz256_add

- Verify the ecp_nistz256_add x86_64 implementation in OpenSSL

  the program computes r from a and b such that $r \equiv a + b \pmod{p}$ where

  - $p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff$,
  - each of the inputs and the output is a 256-bit unsigned integer represented by 4 64-bit registers, and
  - $a < p$, $b < p$, and $r < p$

```
mov     (%rsi),%r8
xor     %r13,%r13
mov     0x8(%rsi),%r9
mov     0x10(%rsi),%r10
mov     0x18(%rsi),%r11
add     (%rdx),%r8
adc     0x8(%rdx),%r9
mov     %r8,%rax
adc     0x10(%rdx),%r10
adc     0x18(%rdx),%r11
mov     %r9,%rdx
adc     $0x0,%r13
sub     (%rsi),%r8
mov     %r10,%rcx
sbb     0x8(%rsi),%r9
sbb     0x10(%rsi),%r10
mov     %r11,%r12
sbb     0x18(%rsi),%r11
sbb     $0x0,%r13
cmovb   %rax,%r8
cmovb   %rdx,%r9
mov     %r8,(%rdi)
cmovb   %rcx,%r10
mov     %r9,0x8(%rdi)
cmovb   %r12,%r11
mov     %r10,0x10(%rdi)
mov     %r11,0x18(%rdi)
```

Z3Py Exercise - ecp_nistz256_add (cont'd)

the program computes r from a and b such that $r \equiv a + b \pmod{p}$ where

- $p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff$,
- each of the inputs and the output is a 256-bit unsigned integer represented by 4 64-bit registers, and
- $a < p$, $b < p$, and $r < p$

rewrite for better readability

```
r8 = a0
r13 = 0
r9 = a1
r10 = a2
r11 = a3
carry r8 = r8 + b0
carry r9 = r9 + b1 + carry
rax = r8
carry r10 = r10 + b2 + carry
carry r11 = r11 + b3 + carry
rdx = r9
r13 = r13 + 0 + carry
carry r8 = r8 - p0
rcx = r10
carry r9 = r9 - p1 - carry
carry r10 = r10 - p2 - carry
r12 = r11
carry r11 = r11 - p3 - carry
carry r13 = r13 - 0 - carry
r8 = carry ? rax : r8
r9 = carry ? rdx : r9
r0 = r8
r10 = carry ? rcx : r10
r1 = r9
r11 = carry ? r12 : r11
r2 = r10
r3 = r11
```

Z3Py Exercise - ecp_nistz256_add (cont'd)

the program computes r from a and b such that $r \equiv a + b \pmod{p}$ where

- $p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff$,
- each of the inputs and the output is a 256-bit unsigned integer represented by 4 64-bit registers, and
- $a < p$, $b < p$, and $r < p$

translate to SSA

```
r8_1 = a0
r13_1 = 0
r9_1 = a1
r10_1 = a2
r11_1 = a3
carry_1 r8_2 = r8_1 + b0
carry_2 r9_2 = r9_1 + b1 + carry_1
rax_1 = r8_2
carry_3 r10_2 = r10_1 + b2 + carry_2
carry_4 r11_2 = r11_1 + b3 + carry_3
rdx_1 = r9_2
r13_2 = r13_1 + 0 + carry_4
carry_5 r8_3 = r8_2 - p0
rcx_1 = r10_2
carry_6 r9_3 = r9_2 - p1 - carry_5
carry_7 r10_3 = r10_2 - p2 - carry_6
r12_1 = r11_2
carry_8 r11_3 = r11_2 - p3 - carry_7
carry_9 r13_3 = r13_2 - 0 - carry_8
r8_4 = carry_9 ? rax_1 : r8_3
r9_4 = carry_9 ? rdx_1 : r9_3
r0 = r8_4
r10_4 = carry_9 ? rcx_1 : r10_3
r1 = r9_4
r11_4 = carry_9 ? r12_1 : r11_3
r2 = r10_4
r3 = r11_4
```