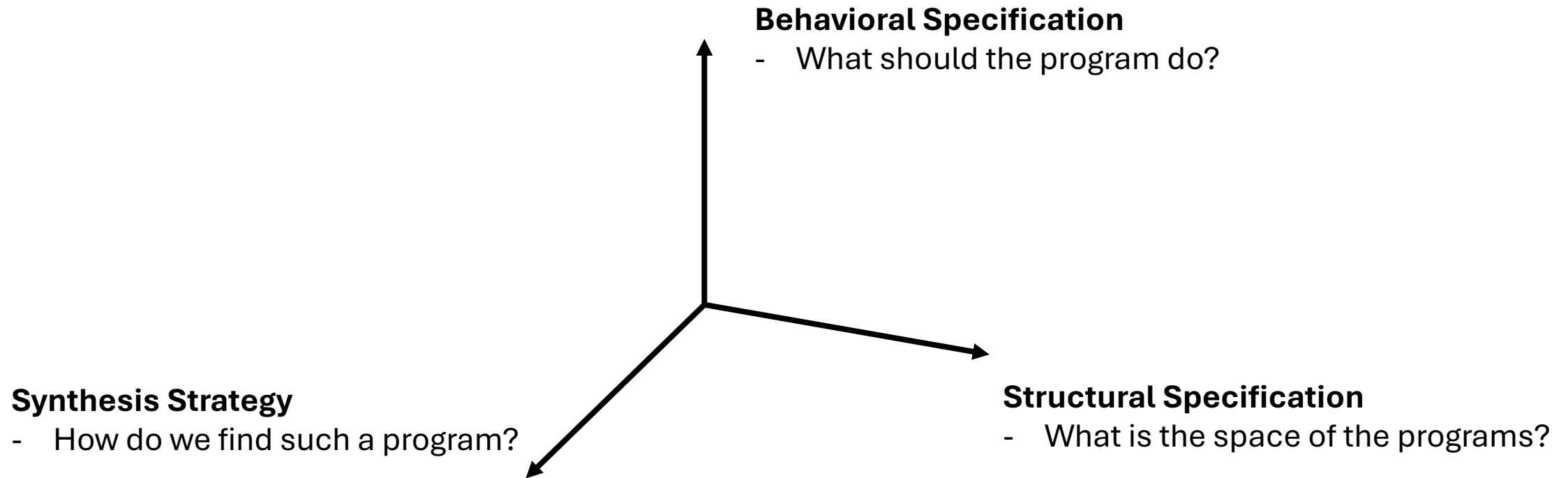


Machine Programming

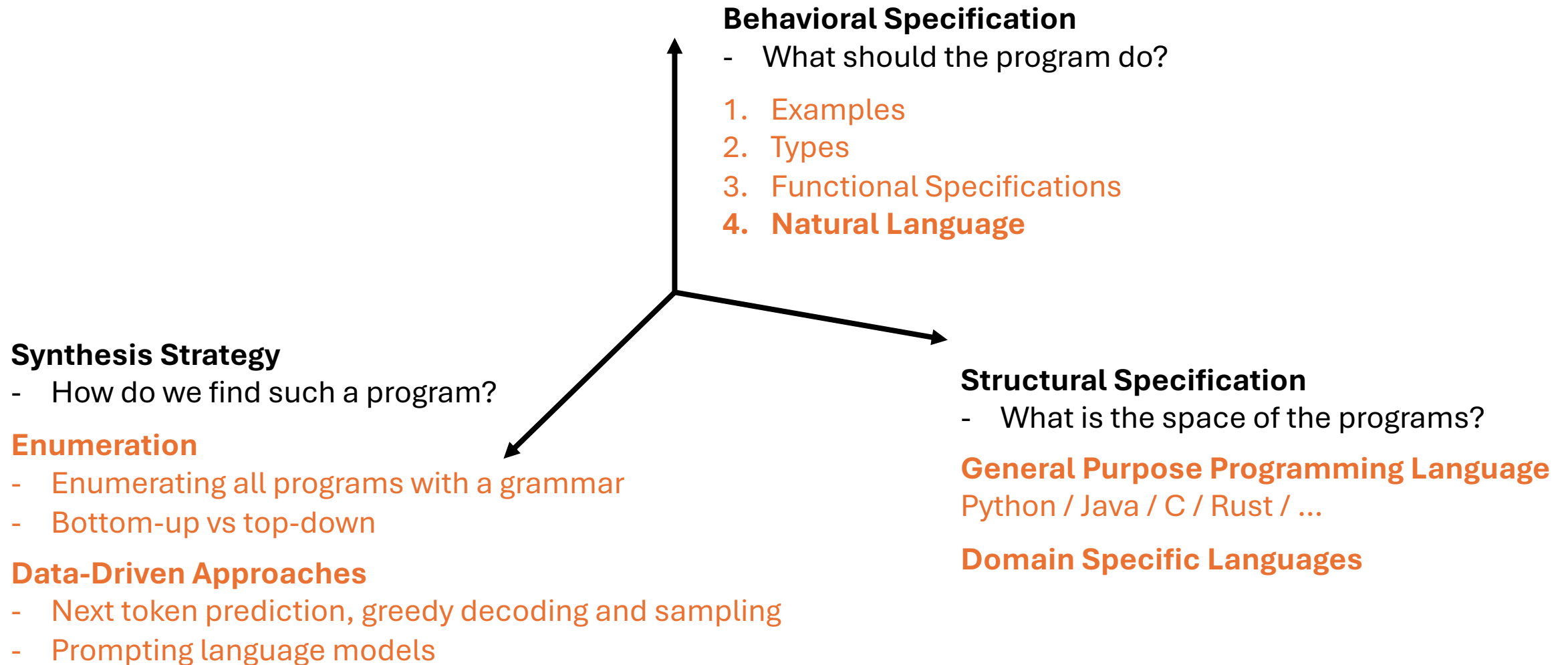
Lecture 7 – Iterative Refinement with Feedback

Ziyang Li

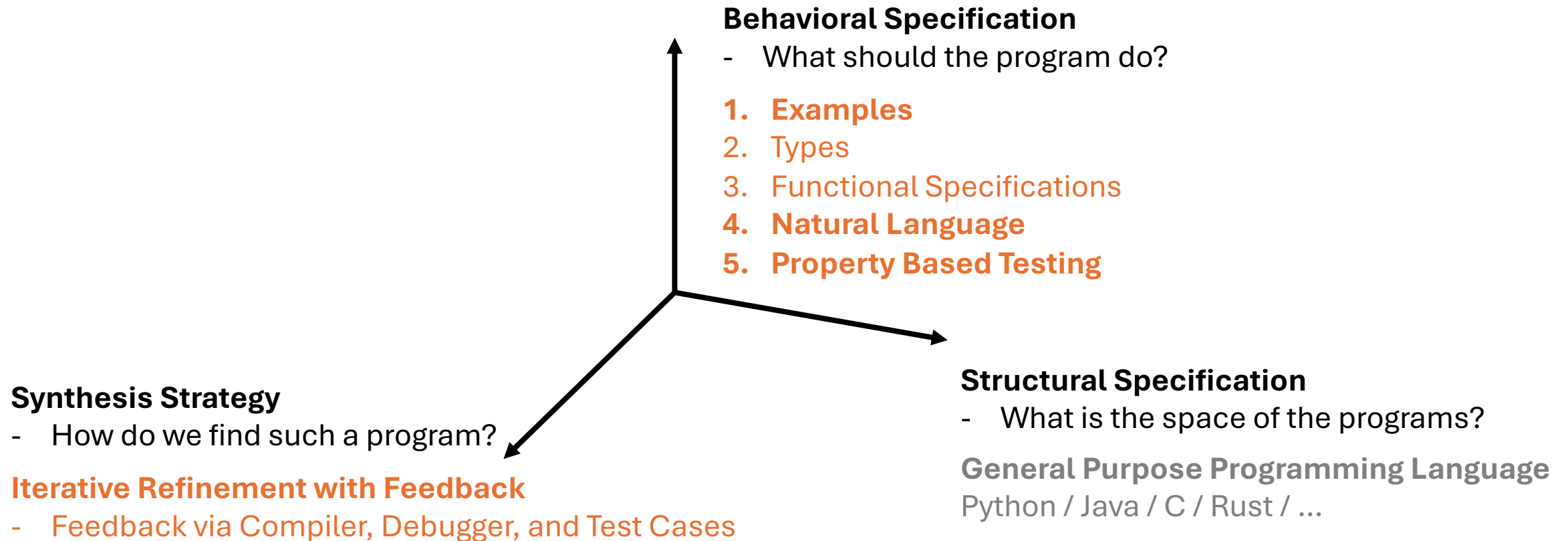
Dimensions in Program Synthesis



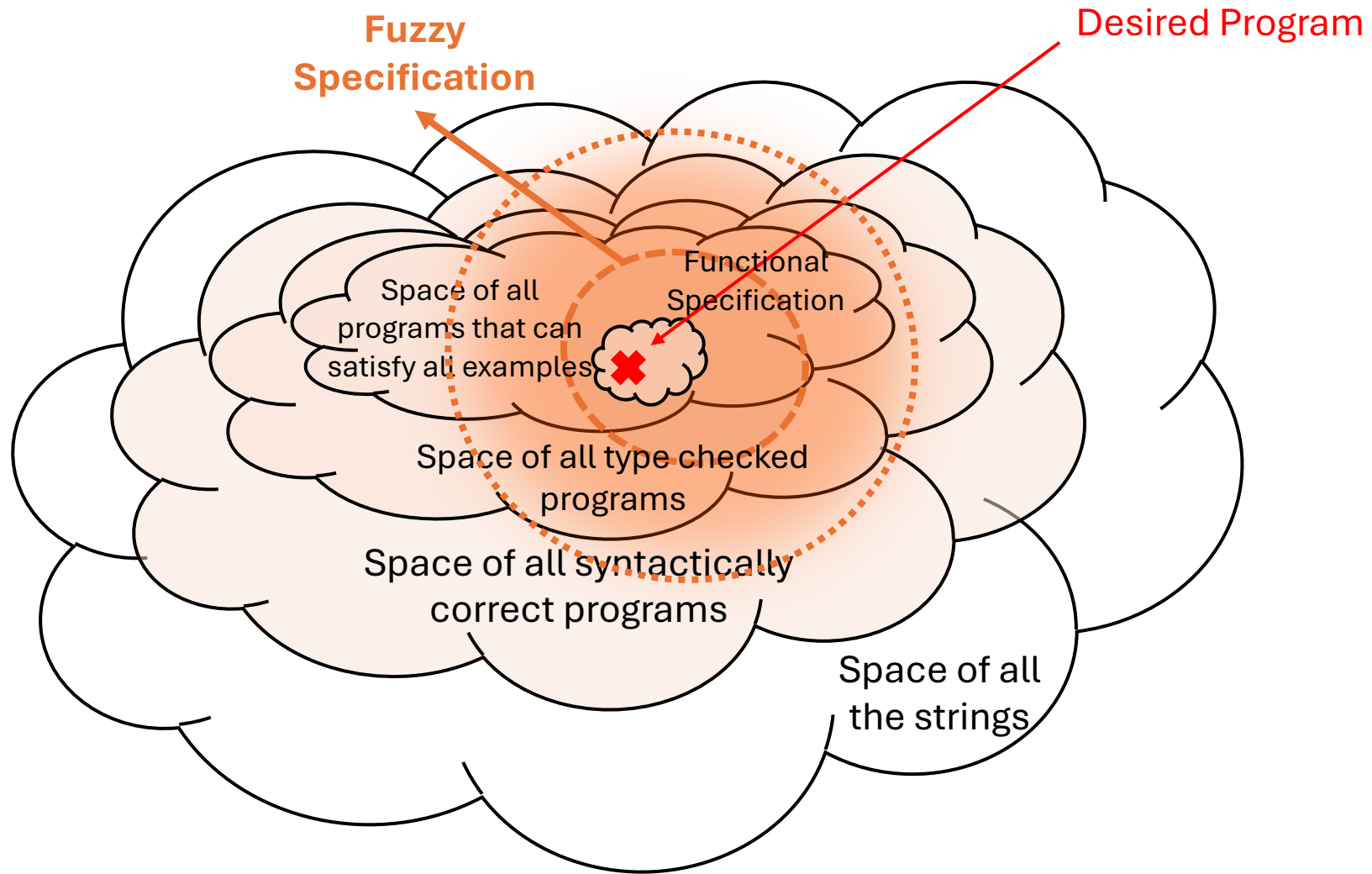
The Course So Far



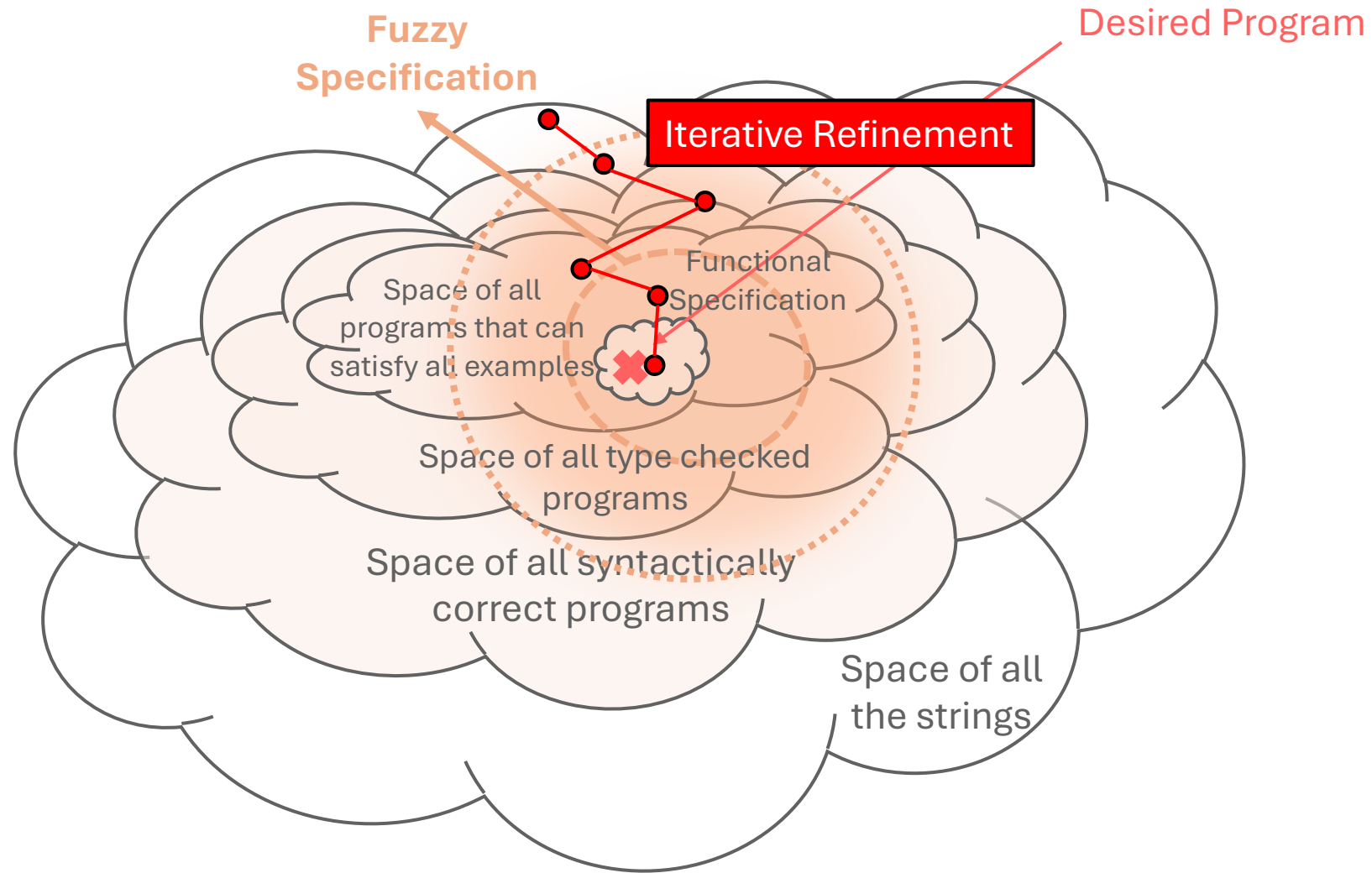
Today



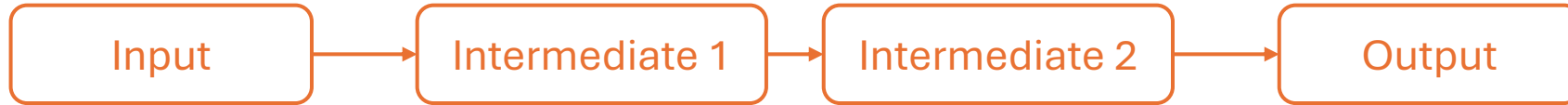
High Level Picture



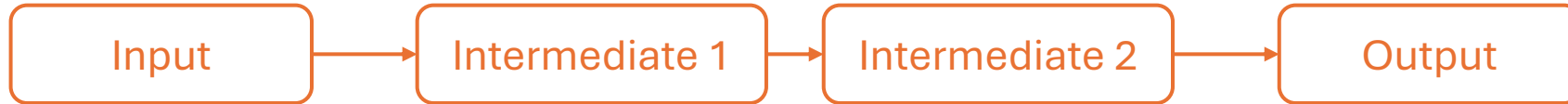
High Level Picture



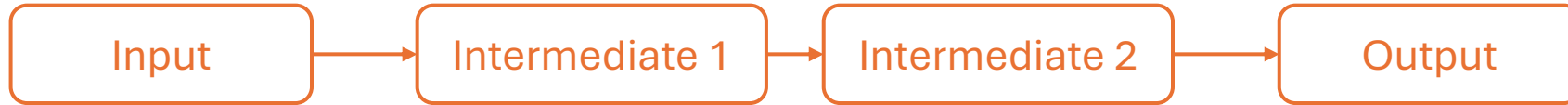
From Multi-Step Prompting to Iterative Prompting



From Multi-Step Prompting to Iterative Prompting



From Multi-Step Prompting to Iterative Prompting

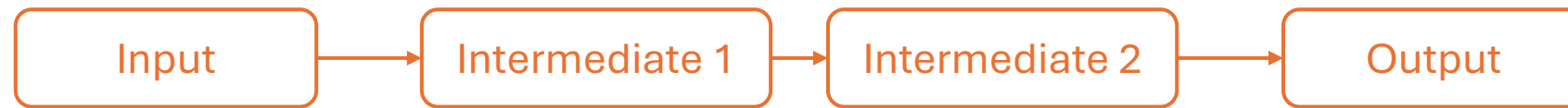


A known and fixed number of steps



A variable number of steps

From Multi-Step Prompting to Iterative Prompting



A known and fixed number of steps

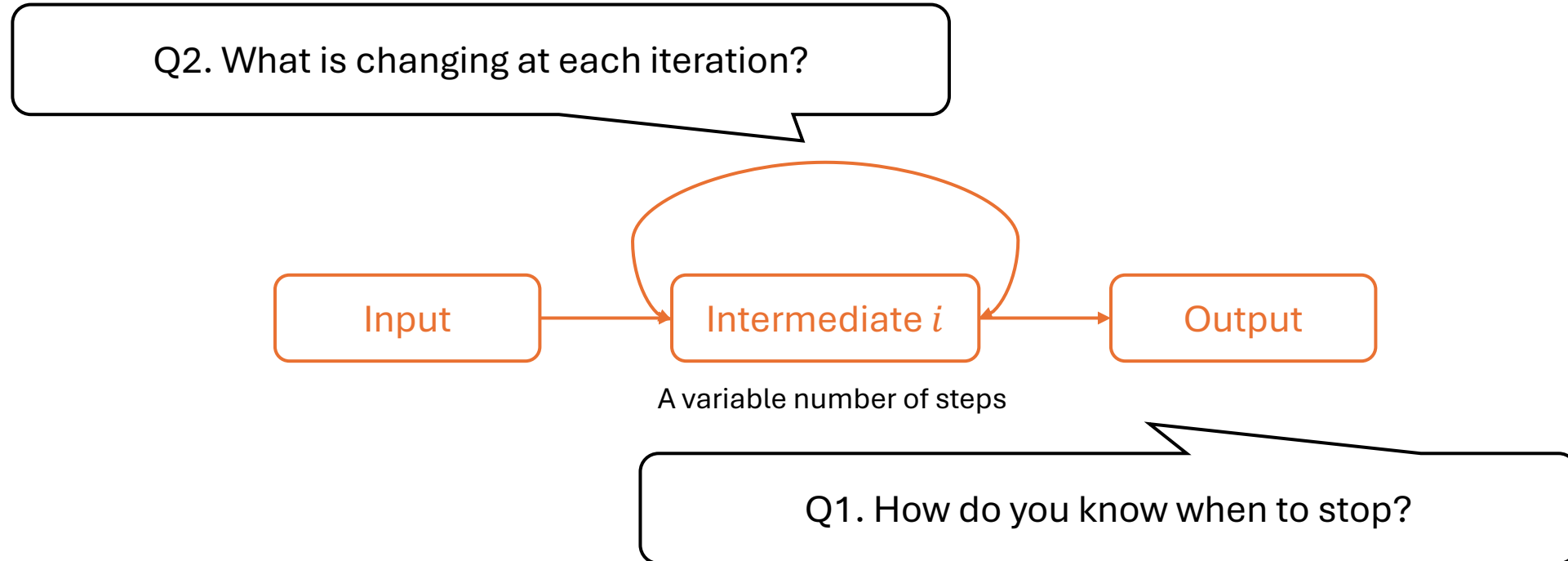


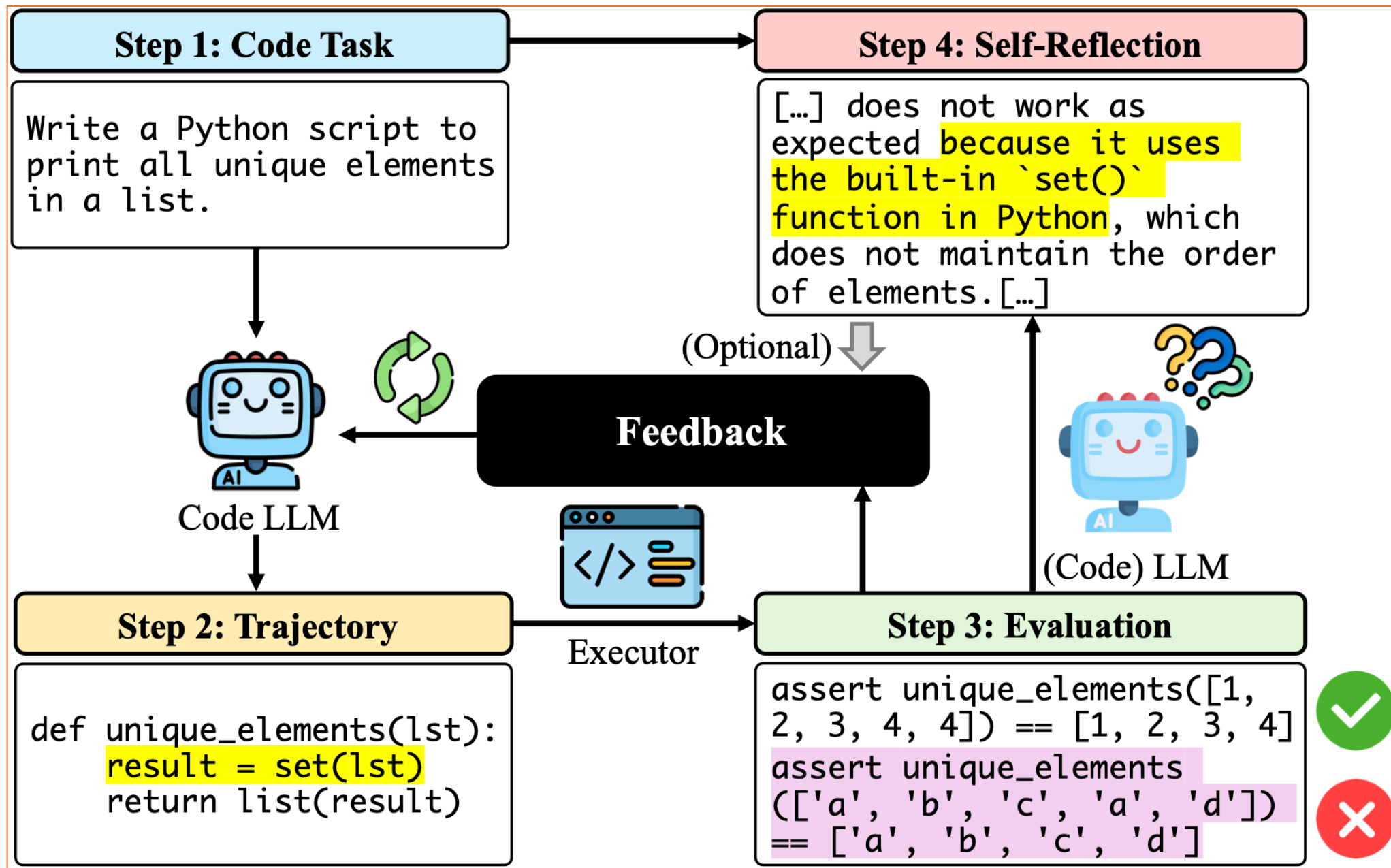
A variable number of steps

- Zero-shot (0)
- Few-shot (0)
- Chain-of-thought (1)
- Self-Reflection (1)
- Transpilation (1)

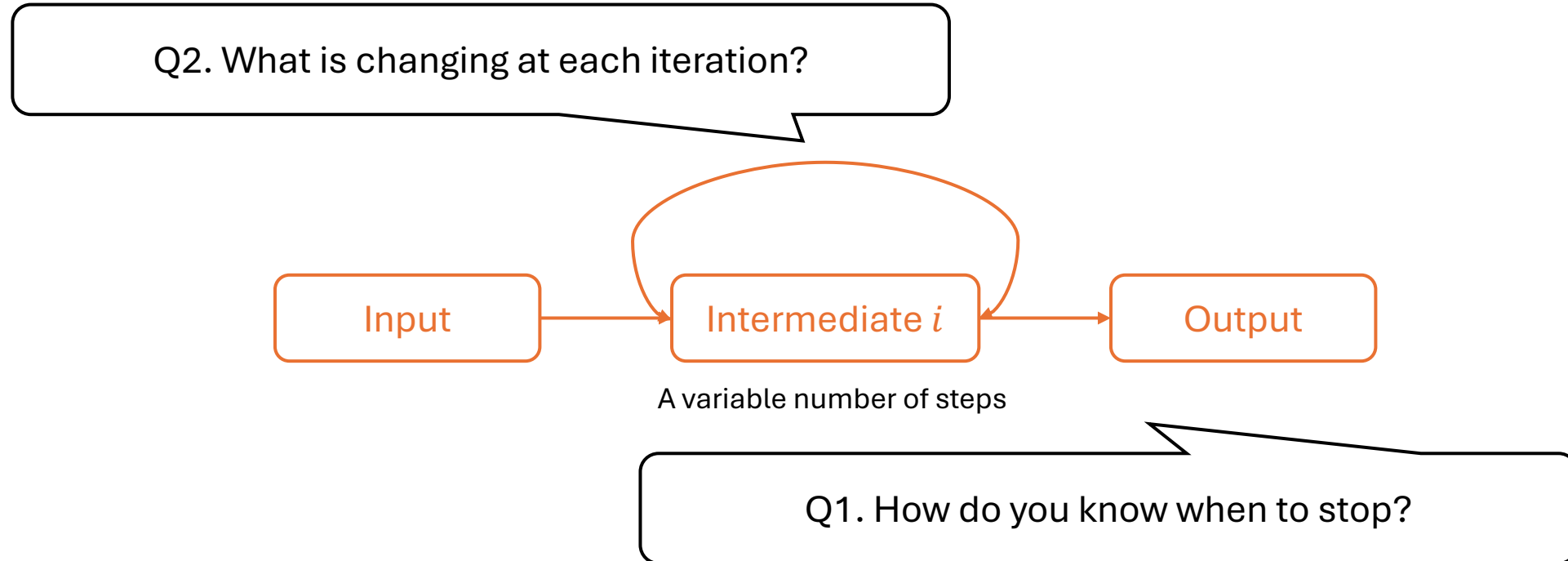
- Self-Debug
- Self-Repair
- FunSearch
- Evolutionary Search
- Generational Search

From Multi-Step Prompting to Iterative Prompting

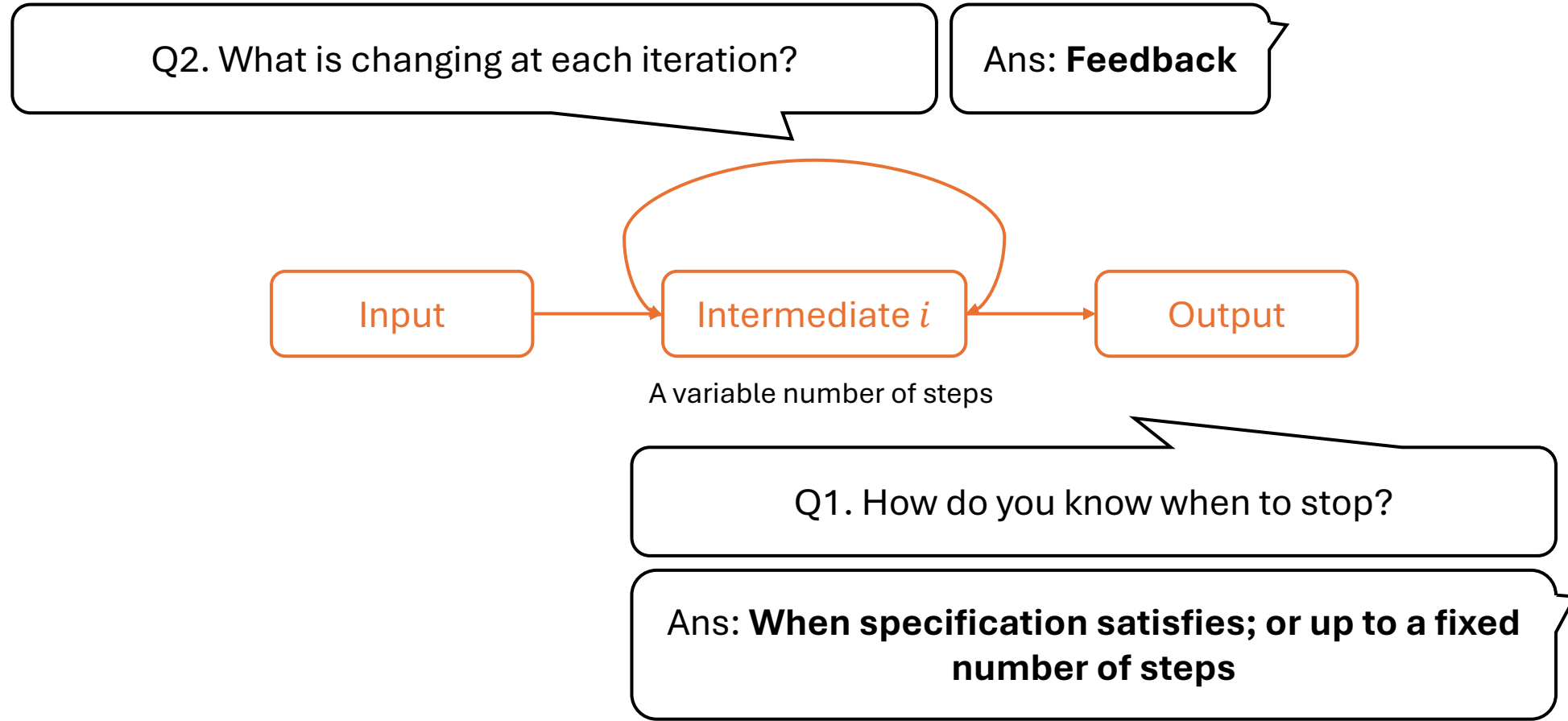




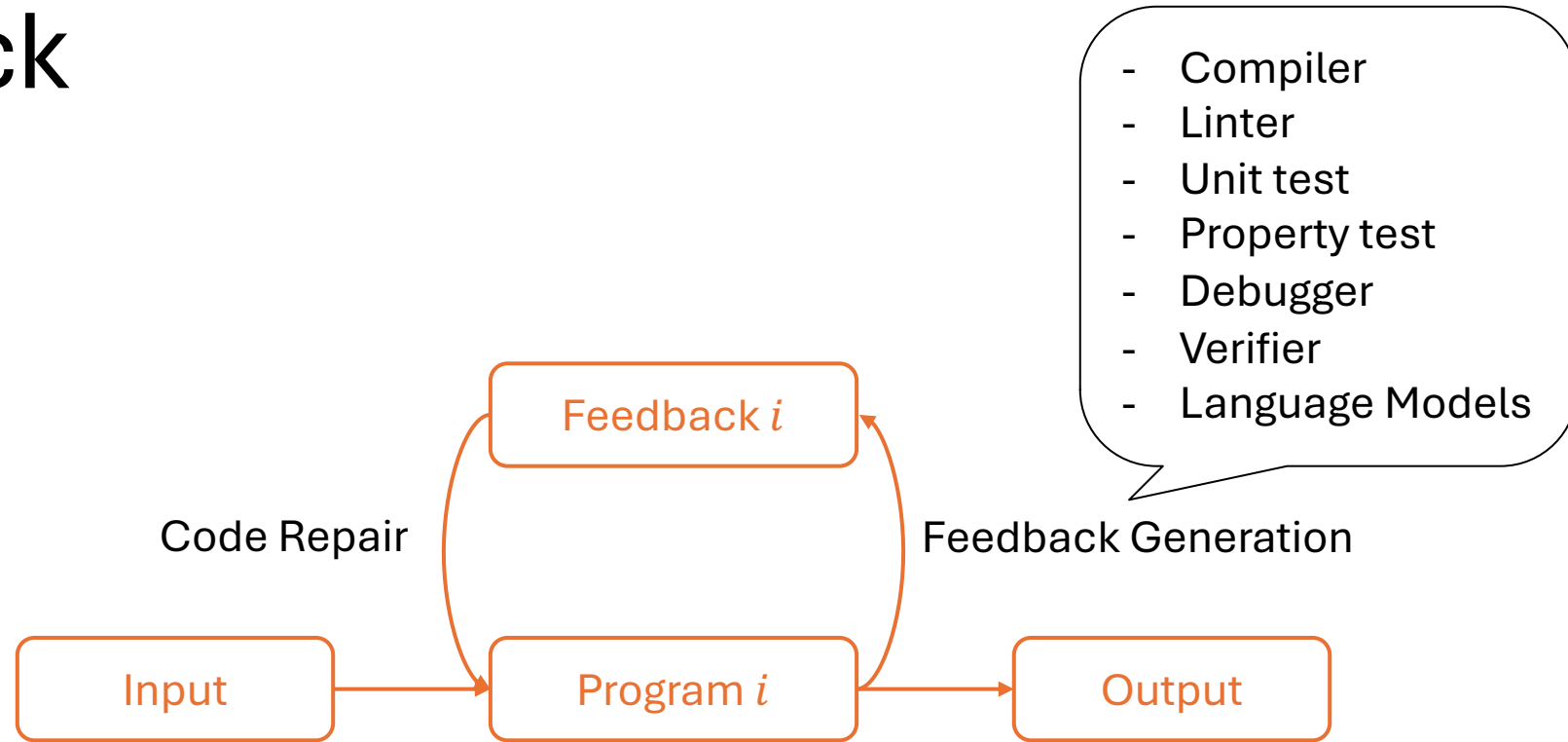
From Multi-Step Prompting to Iterative Prompting

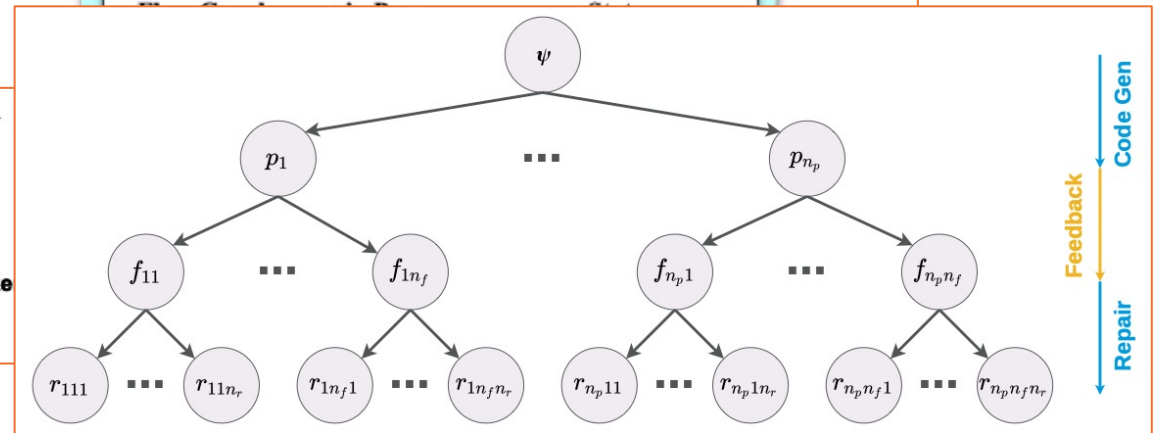
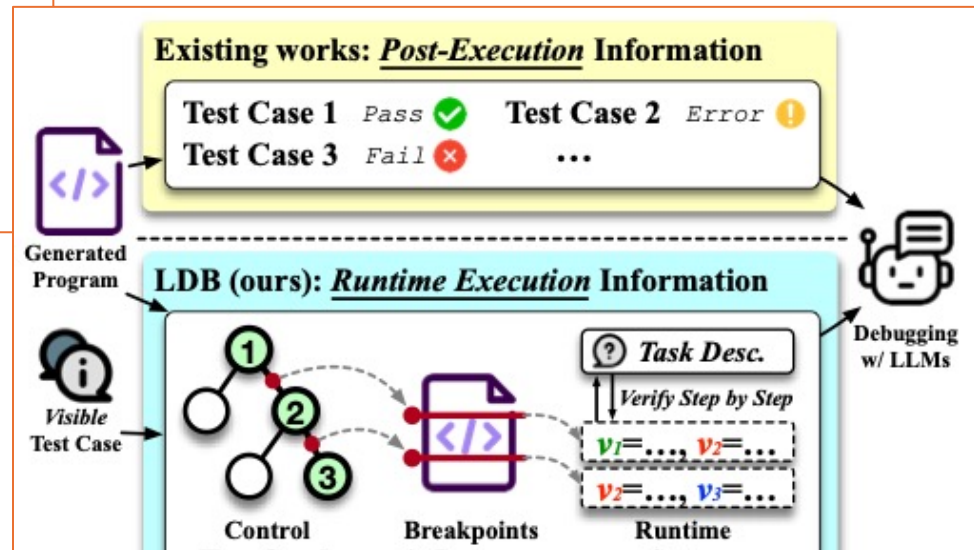
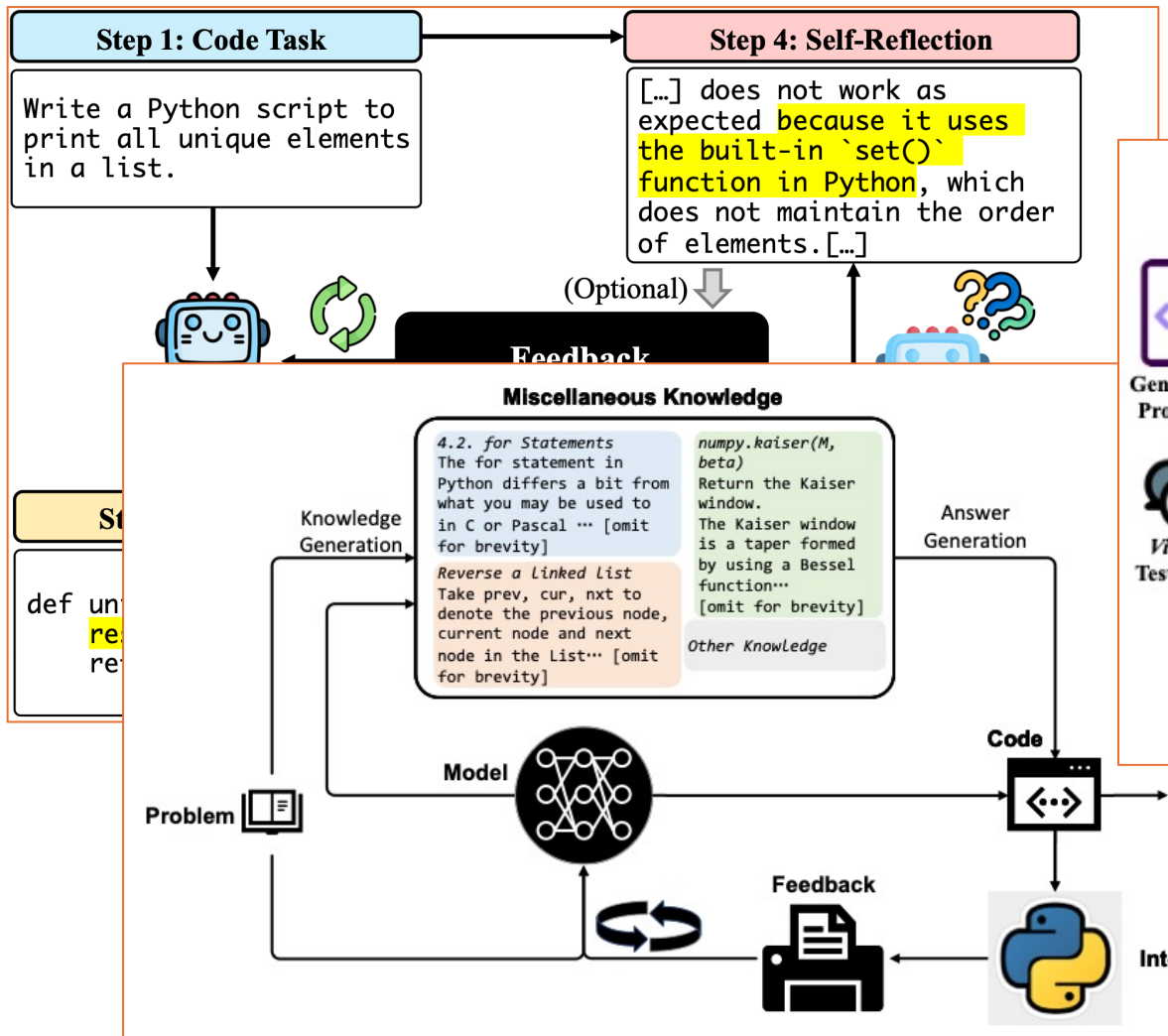


From Multi-Step Prompting to Iterative Prompting

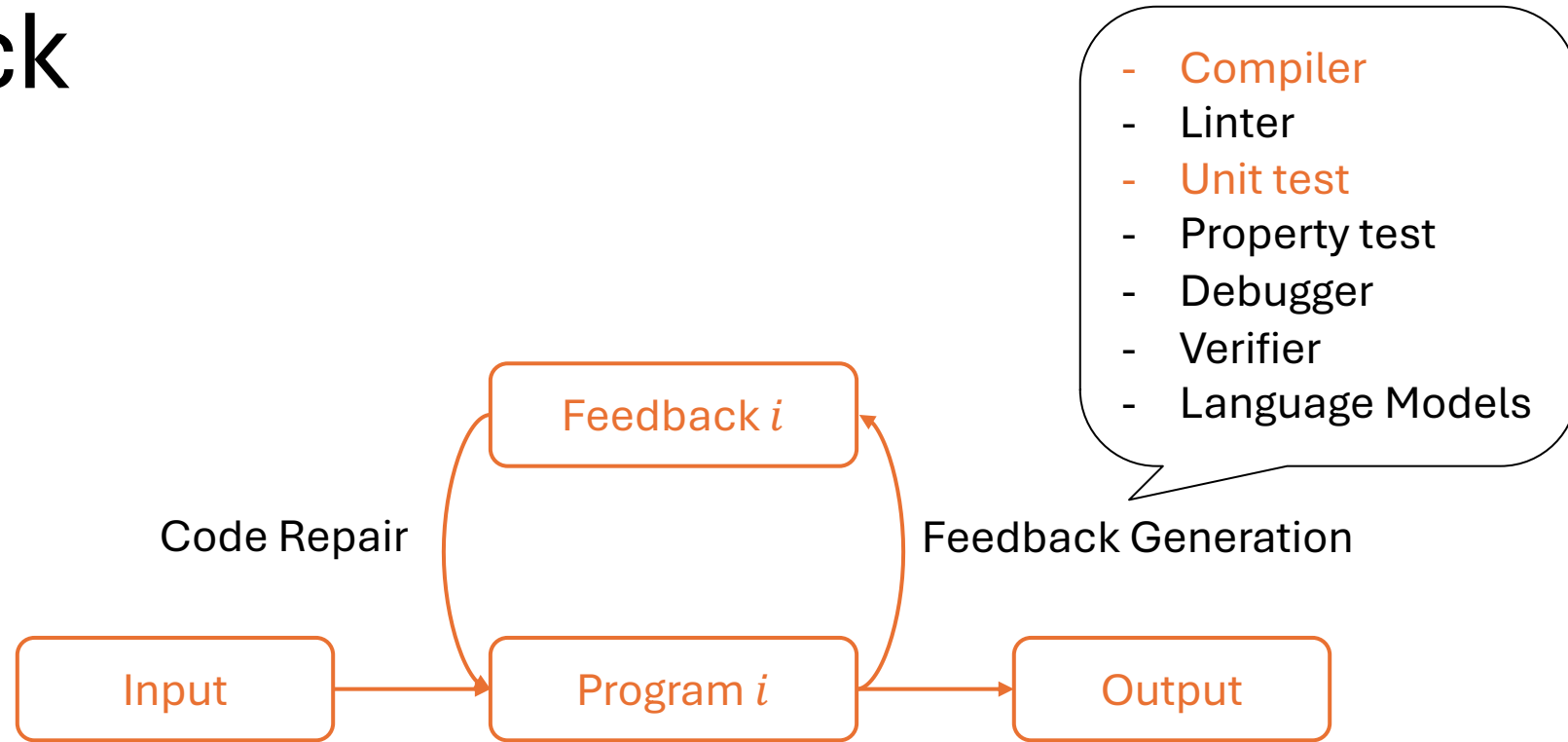


Feedback





Feedback



Compiler Feedback

- Classes of feedback
 - Syntax errors
 - Type checking errors
 - Language specific errors (type cast, memory, lifetime, etc.)
- Issues of too many errors
 - Verbose errors from lexer/parser/linter
 - Root error cascades and induce more errors
- Solutions
 - Heuristics: feed first few errors first!
 - Find root cause
 - Group errors by files

```
expression of type 'int' [-Wint-conversion]
30 | char *not_a_ptr = 42;           // error: invalid conversion
    | ^~~~~
a.c:31:6: warning: incompatible pointer types initializing 'int **' with an expression of type 'int **'; remove & [-Wincompatible-pointer-types]
31 | int *ptr_to_self = &ptr_to_self; // error: invalid initializer
    | ^~~~~
a.c:35:6: error: conflicting types for 'f'
35 | void f(double);                 // error: conflicting types for 'f'
    | ^
a.c:34:6: note: previous declaration is here
34 | void f(int);
    | ^
a.c:36:5: error: conflicting types for 'f'
36 | int f(int x) { return "nope"; } // error: returning wrong type (string to int)
    | ^
a.c:34:6: note: previous declaration is here
34 | void f(int);
    | ^
a.c:36:23: error: incompatible pointer to integer conversion returning 'char[5]' from a function with result type 'int' [-Wint-conversion]
36 | int f(int x) { return "nope"; } // error: returning wrong type (string to int)
    | ^~~~~
a.c:38:5: warning: a function declaration without a prototype is deprecated in all versions of C and is treated as a zero-parameter prototype in C23, conflicting with a subsequent definition [-Wdeprecated-non-prototype]
38 | int g();                        // old-style declaration
    | ^
a.c:39:5: note: conflicting prototype is here
39 | int g(int x) { return x; }      // error: conflicting types for 'g'
    | ^
a.c:43:5: error: 'break' statement not in loop or switch statement
43 | break;                          // error: 'break' not in loop or switch
    | ^
a.c:47:5: error: 'continue' statement not in loop statement
47 | continue;                       // error: 'continue' not in loop
    | ^
a.c:51:5: error: 'case' statement not in switch statement
51 | case 3: printf("nope\n");       // error: 'case' label not within a switch statement
    | ^
a.c:66:1: error: redefinition of label 'label'
66 | label:                          // error: duplicate label
    | ^
a.c:64:1: note: previous definition is here
64 | label:
    | ^
a.c:71:36: error: argument may not have 'void' type
71 | void nonsense_params(register void v) { // error: 'void' parameter variable
    | ^
a.c:77:10: error: expected ';' at end of declaration list
77 | int x
    | ^
fatal error: too many errors emitted, stopping now [-ferror-limit=]
4 warnings and 20 errors generated.
```

Linter Feedback

- What is a Linter
 - Programming errors and bugs
 - Stylistic inconsistencies
 - Suspicious constructs or “code smells”
 - Potential security vulnerabilities
- Catch
 - Usually very local
 - Many “false positives”
 - Most linter errors do not affect execution

```
liby99@hanabi ~/L/P/m/lecture-7-demos> npx eslint bad.js
/Users/liby99/Local/Projects/machine-programming/lecture-7-demos/bad.js
   4:1  error  'foo' is not defined
no-undef
   7:5  error  'neverChanges' is assigned a value but never used
no-unused-vars
  10:7  error  'greeting' is assigned a value but never used
no-unused-vars
  13:14 error  'foo' is not defined
no-undef
  13:20 error  'console' is not defined
no-undef
  16:10 error  'doStuff' is defined but never used
no-unused-vars
  17:3  error  'console' is not defined
no-undef
  18:17 error  Empty block statement
no-empty
  20:3  error  Unreachable code
no-unreachable
  20:3  error  'console' is not defined
no-undef
  24:21 error  Duplicate key 'a'
no-dupe-keys
  25:9  error  Do not access Object.prototype method 'hasOwnProperty' from
no-prototype-builtins
  26:3  error  'console' is not defined
no-undef
  30:10 error  'shadow' is defined but never used
no-unused-vars
  31:7  error  'x' is already defined
no-redeclare
  36:10 error  'parseStuff' is defined but never used
no-unused-vars
  41:7  error  'wrappedStr' is assigned a value but never used
no-unused-vars
  42:7  error  'wrappedNum' is assigned a value but never used
no-unused-vars
  45:1  error  Unexpected newline between object and [ of property access
no-unexpected-multiline
```

Linter Feedback

- What is a Linter
 - Programming errors and bugs
 - Stylistic inconsistencies
 - Suspicious constructs or “code smells”
 - Potential security vulnerabilities
- Catch
 - Usually very local
 - Many “false positives”
 - Most linter errors do not affect execution
 - Should be careful when presented to a language model

Help me write a javascript function `parity(x)` to find the parity of a given number.

Sure! In number theory / computer science, the **parity** of an integer usually means whether it has an **even** or **odd** number of **1** bits in its binary representation. Sometimes people just mean “is the number itself even or odd.” I’ll show you both:

```
JS parity.js > ...
1  function parity(x) {
2      let result = 0;
3      while (x > 0) {
4          result ^= (x & 1); // XOR with least significant bit
5          x >>= 1;           // shift right
6      }
7      return result; // 0 for even parity, 1 for odd parity
8  }
9
10 // Examples
11 console.log(parity(7)); // 1 → because 7 (111b) has three 1s
12 console.log(parity(12)); // 0 → because 12 (1100b) has two 1s
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos> npx eslint parity.js
```

```
/Users/liby99/Local/Projects/machine-programming/lecture-7-demos/parity.js
11:1  error  'console' is not defined  no-undef
12:1  error  'console' is not defined  no-undef
```

```
* 2 problems (2 errors, 0 warnings)
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos [1]> node parity.js
```

```
1
0
```

False positives

Unit Test Feedback

- Unit Tests
 - Encode basic input-output pairs
 - Usually provided along with the synthesis task as a part of behavioral specifications
- Encoding
 - As assertion
 - Using unit test libraries

```
parity_buggy.py > ...
1  def parity(x: int) -> int:
2      if x == 0:
3          return 1 # BUG
4
5      res = 0
6      while x:
7          res ^= (x & 1)
8          x >>= 1
9      return res
10
11 # ----- Tests (simple assertions) -----
12 assert parity(0) == 0, "parity(0) should be 0 (even), but
13 assert parity(1) == 1, "1 -> 1 (binary), odd -> 1"
14 assert parity(2) == 1, "2 -> 10 (binary), one '1' -> 1"
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos> python3 parity_buggy.py
Traceback (most recent call last):
  File "/Users/liby99/Local/Projects/machine-programming/lecture-7-demos/parity_buggy.py",
    line 13, in <module>
      assert parity(0) == 0, "parity(0) should be 0 (even), but the function returns 1"
      ~~~~~
AssertionError: parity(0) should be 0 (even), but the function returns 1
```

Unit Test Feedback

- Unit Tests
 - Encode basic input-output pairs
 - Usually provided along with the synthesis task as a part of behavioral specifications
- Encoding
 - As assertion
 - Using **unit test libraries**

```
class TestParity(unittest.TestCase):  
    def test_zero_even_parity(self):  
        # Fails with the buggy implementation  
        self.assertEqual(parity(0), 0)
```

Python (using unittest)

```
#[test]  
fn zero_has_even_parity() {  
    // Fails with the buggy implementation  
    assert_eq!(parity(0), 0);  
}
```

Rust

```
spec :: Spec  
spec = do  
    describe "parity" $ do  
        it "returns 0 for zero (even parity)" $ do  
            parity 0 `shouldBe` 0
```

Haskell (using Test.Hspec)

```
TEST(Parity, ExamplesTable) {  
    struct Case { uint64_t x; int expected; };  
    Case cases[] = { {1, 1}, {2, 1}, {3, 0}, {7, 1} };  
    for (auto c : cases) {  
        EXPECT_EQ(parity(c.x), c.expected) << "x = " << c.x;  
    }  
}
```

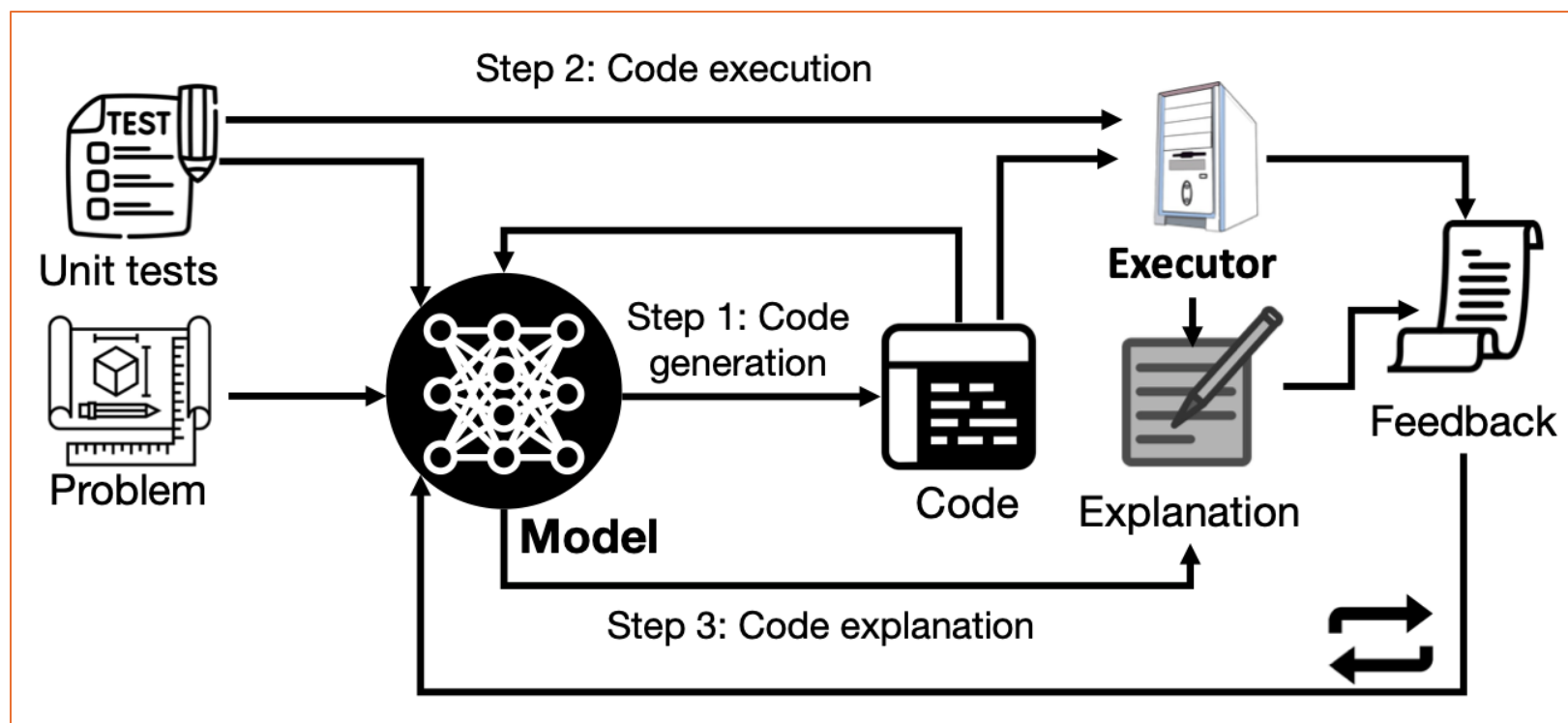
C++ (using GoogleTest)

TEACHING LARGE LANGUAGE MODELS TO SELF-DEBUG

Xinyun Chen¹ Maxwell Lin² Nathanael Schärli¹ Denny Zhou¹

¹ Google DeepMind ² UC Berkeley

{xinyunchen,schaerli,dennyzhou}@google.com, mxlin@berkeley.edu



TEACHING LARGE LANGUAGE MODELS TO SELF-DEBUG

Xinyun Chen¹ Maxwell Lin² Nathanael Schärli¹ Denny Zhou¹

¹ Google DeepMind ² UC Berkeley

{xinyunchen,schaerli,dennyzhou}@google.com, mxlin@berkeley.edu

Simple Feedback	Unit Test (UT) Feedback	Unit Test + Explanation (+Expl.)	Unit Test + Trace (+Trace)
Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.	Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.	Below are C++ programs with incorrect Python translations. <i>Explain the original code, then explain the translations line by line</i> and correct them using the provided feedback.	Below are C++ programs with incorrect Python translations. Using the provided feedback, <i>trace through the execution of the translations to determine what needs to be fixed</i> , and correct the translations.
[Original Python] [C++]	[Original Python] [C++]	[Original Python] [C++]	[Original Python] [C++]
[Simple Feedback]	[UT Feedback]	[C++ Explanation] [UT Feedback]	[Trace] [UT Feedback]
[Revised Python #1]	[Revised Python #1]	[Python Explanation] [UT Feedback]	[Revised Python #1] [UT Feedback]
[Simple Feedback]	[UT Feedback]	[Revised Python #1] [Python Explanation] [UT Feedback]	[Trace] [Revised Python #2] [UT Feedback]
[Revised Python #2]	[Revised Python #2]	[Revised Python #2] [Python Explanation]	...
...	...	...	...

[Original Python] =

```
def count_trailing_zeroes_factorial_number(n):  
    cnt = 0  
    i = 5  
    while ((n / i) >= 1):  
        cnt = (cnt + (n / i))  
        i *= 5  
    return cnt
```

[Simple Feedback] =

The above Python translation does not do the same thing as the C++ code. Correct the Python translation.

[UT Feedback] =

The Python translation does not do the same thing as the C++ code. These are the results of failed unit tests that test whether the Python translation's outputs match the C++ program's outputs:

Failed: assert

count_trailing_zeroes_factorial_number(9) == 1

Actual Result: 1.8

Correct the translation.

[Revised Python #n] =

```
def count_trailing_zeroes_factorial_number(n):  
    cnt = 0  
    i = 5  
    while ((n / i) >= 1):  
        cnt = (cnt + (n // i))  
        i *= 5  
    return cnt
```

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	80.6	52.2
+ Expl.	69.8	74.2	80.4	52.2
+ Trace.	70.8	72.8	80.2	53.2

Self-Debugging with Different Feedback Formats

Unit Test Trace Feedback

- Issue
 - Unit test feedback only covers the expected output
 - It does not shed light on the **Root Cause**
 - Demands further analysis
- Solution
 - Reason about program execution via the use of **Debugger**

```
parity_buggy.py > ...
1  def parity(x: int) -> int:
2      if x == 0:
3          return 1 # BUG
4
5      res = 0
6      while x:
7          res ^= (x & 1)
8          x >>= 1
9      return res
10
11 # ----- Tests (simple assertions) -----
12 assert parity(0) == 0, "parity(0) should be 0 (even), but
13 assert parity(1) == 1, "1 -> 1 (binary), odd -> 1"
14 assert parity(2) == 1, "2 -> 10 (binary), one '1' -> 1"
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos> python3 parity_buggy.py
Traceback (most recent call last):
  File "/Users/liby99/Local/Projects/machine-programming/lecture-7-demos/parity_buggy.py",
    line 13, in <module>
      assert parity(0) == 0, "parity(0) should be 0 (even), but the function returns 1"
      ~~~~~^~~~~~
AssertionError: parity(0) should be 0 (even), but the function returns 1
```

Debugger

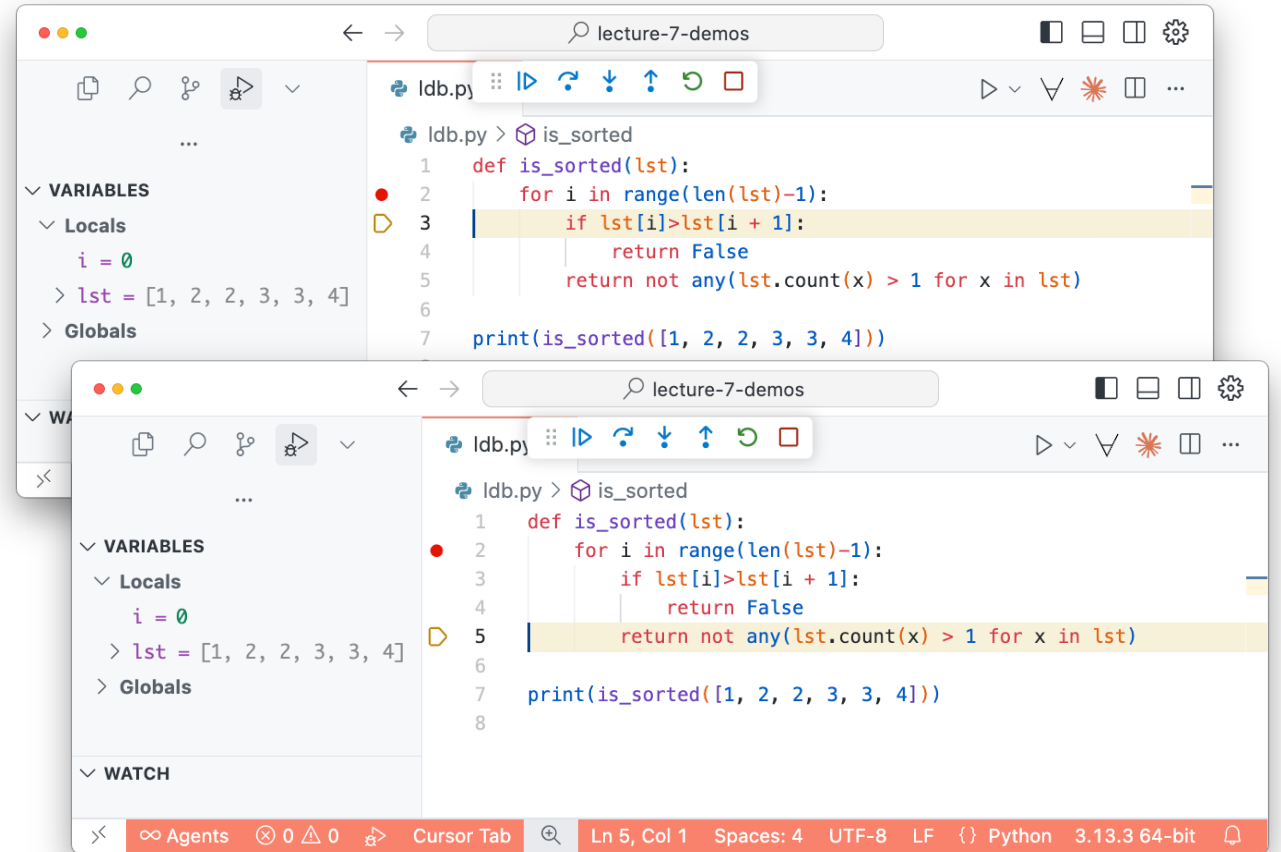
```
def is_sorted(lst):  
    for i in range(len(lst)-1):  
        if lst[i]>lst[i + 1]:  
            return False  
    return not any(lst.count(x) > 1 for x in lst)  
  
print(is_sorted([1, 2, 2, 3, 3, 4]))
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos> python3 ldb.py  
False
```

Debugger

```
def is_sorted(lst):  
    for i in range(len(lst)-1):  
        if lst[i]>lst[i + 1]:  
            return False  
    return not any(lst.count(x) > 1 for x in lst)  
  
print(is_sorted([1, 2, 2, 3, 3, 4]))
```

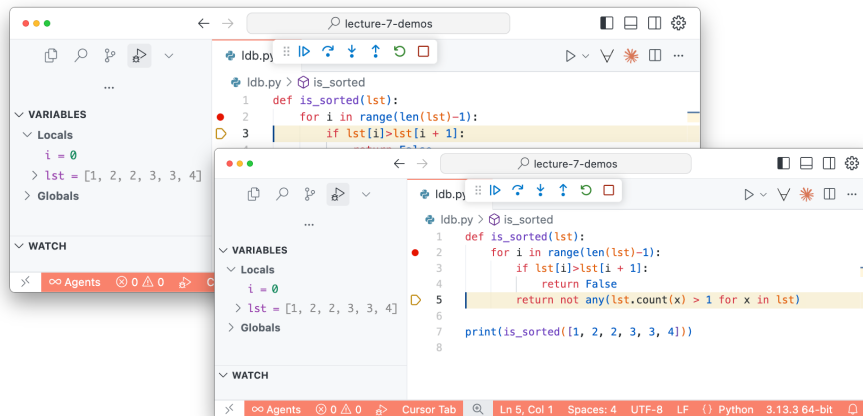
```
liby99@hanabi ~/L/P/m/lecture-7-demos> python3 ldb.py  
False
```



Debugger

```
def is_sorted(lst):  
    for i in range(len(lst)-1):  
        if lst[i]>lst[i + 1]:  
            return False  
    return not any(lst.count(x) > 1 for x in lst)  
  
print(is_sorted([1, 2, 2, 3, 3, 4]))
```

```
liby99@hanabi ~/L/P/m/lecture-7-demos> python3 ldb.py  
False
```



Program State Trace

line 2: `lst = [1, 2, 2, 3, 3, 4]`

line 3: `i = 0`
`lst = [1, 2, 2, 3, 3, 4]`

line 5: `i = 0`
`lst = [1, 2, 2, 3, 3, 4]`
`x = 1`

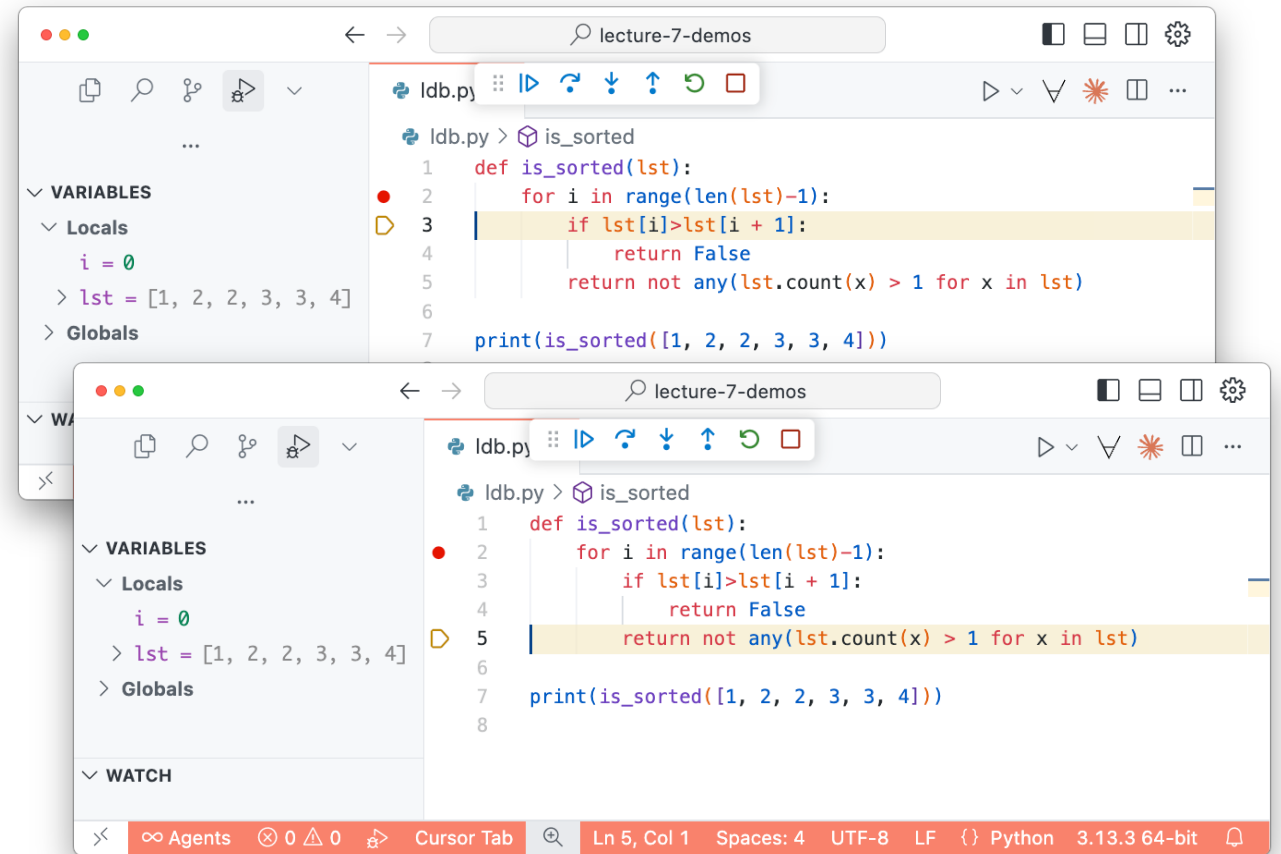
line 6: `c = 1`
`i = 0`
`lst = [1, 2, 2, 3, 3, 4]`
`x = 1`

...

Debugger for Feedback

- Issues

- Too many intermediate states to step through
- Loops may iterate many times resulting in very verbose feedback



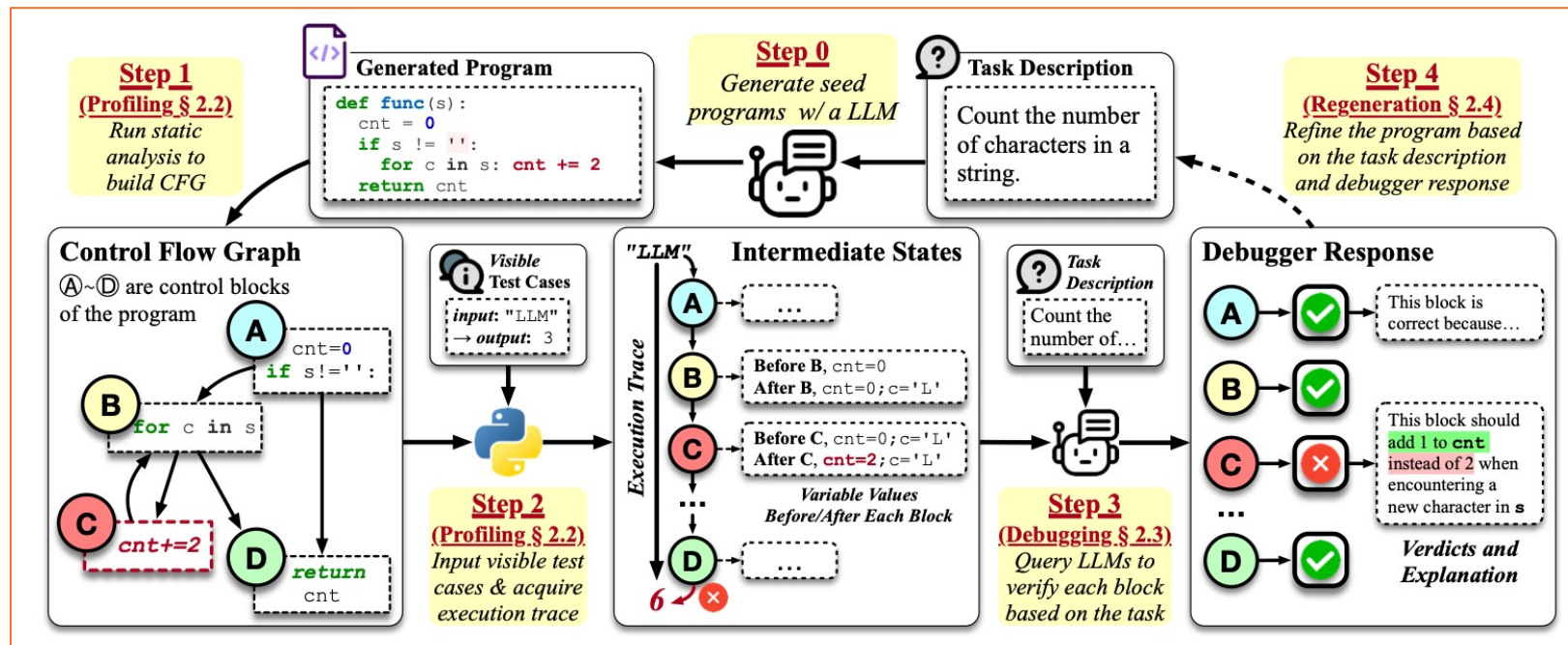
Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step

Li Zhong Zilong Wang[†] Jingbo Shang[†]

University of California, San Diego
{lizhong, zlwang, jshang}@ucsd.edu

Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step

Li Zhong Zilong Wang[†] Jingbo Shang[†]
University of California, San Diego
{lizhong, zlwang, jshang}@ucsd.edu



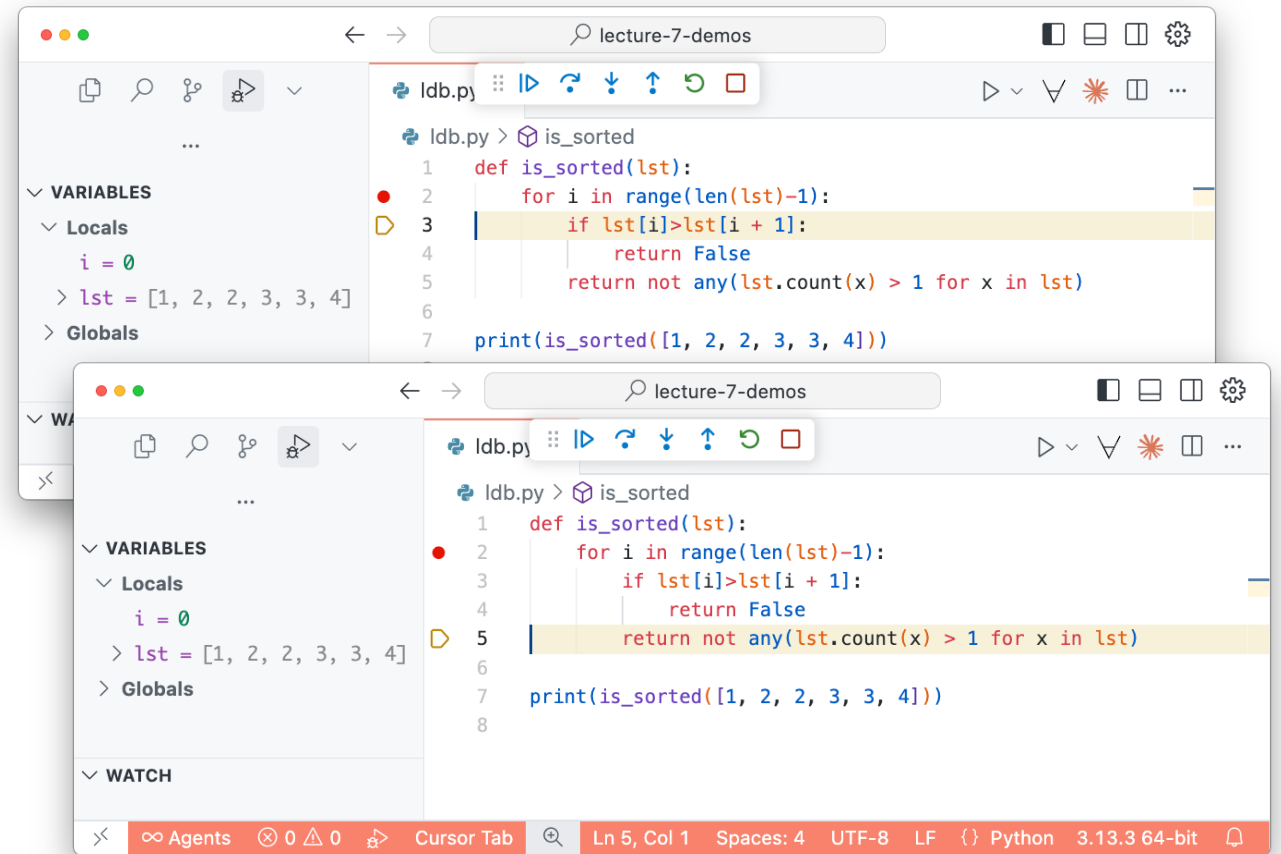
Debugger for Feedback

- Issues

- Too many intermediate states to step through
- Loops may iterate many times resulting in very verbose feedback

- Solution

- Use **Control Flow Graph** to simplify the program trace

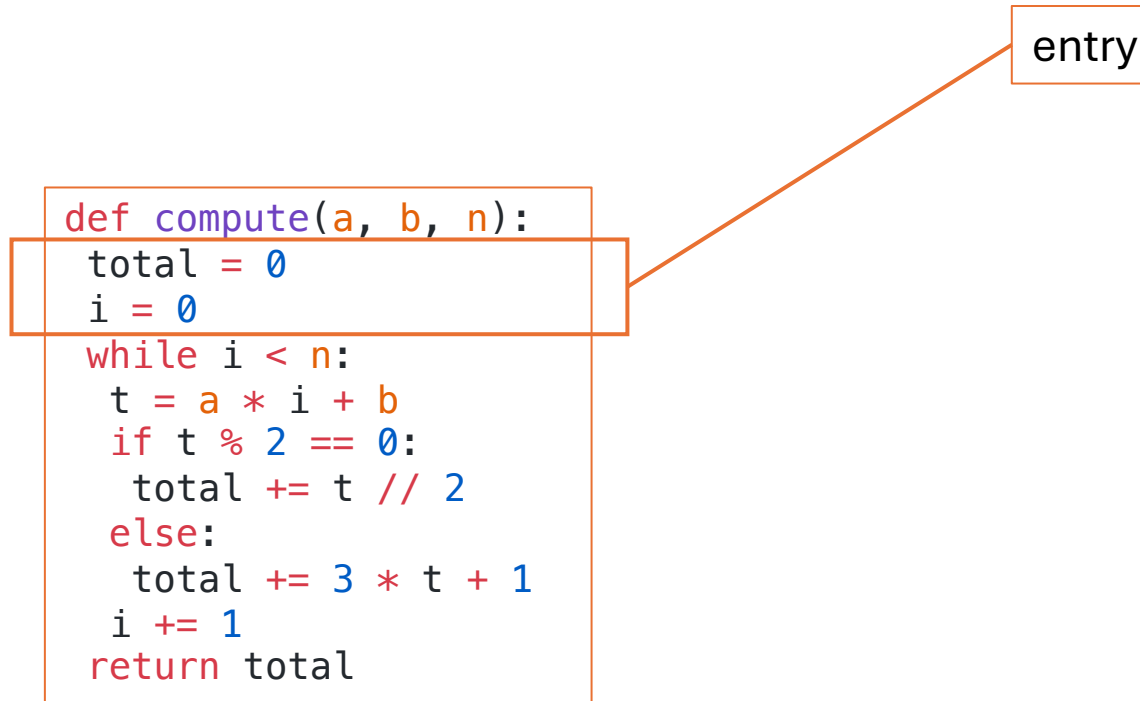


Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

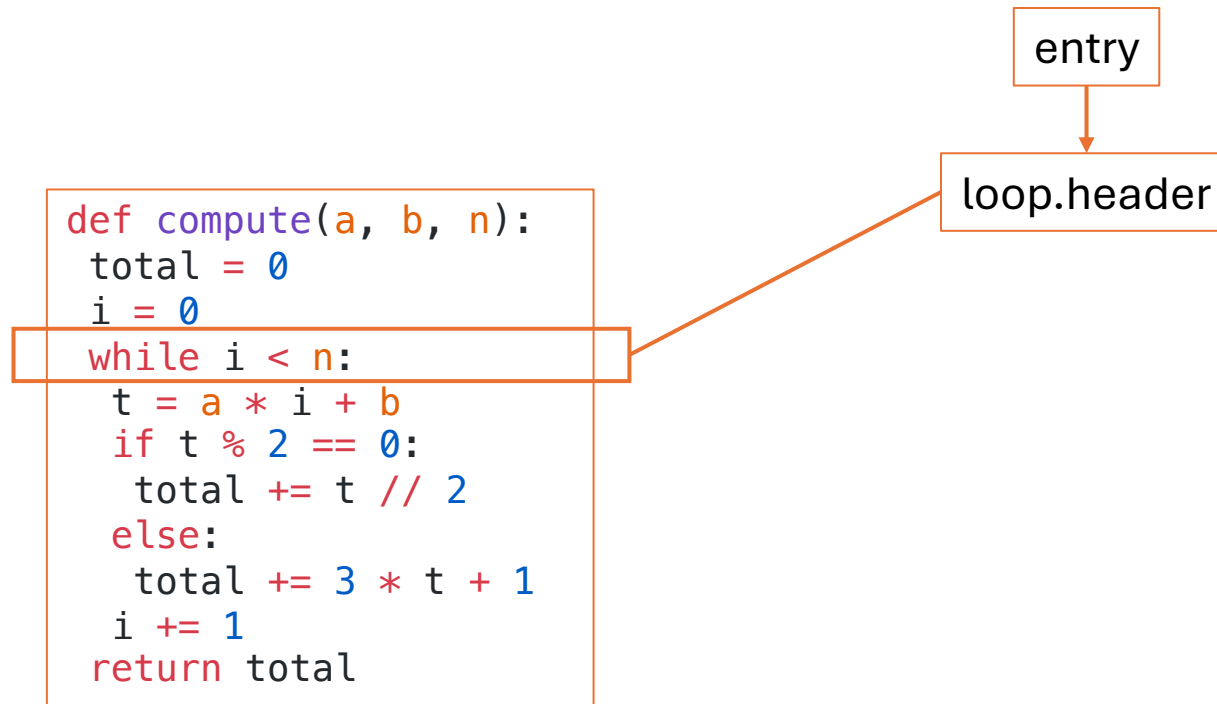
Control Flow Graph



Python Code

Control Flow Graph

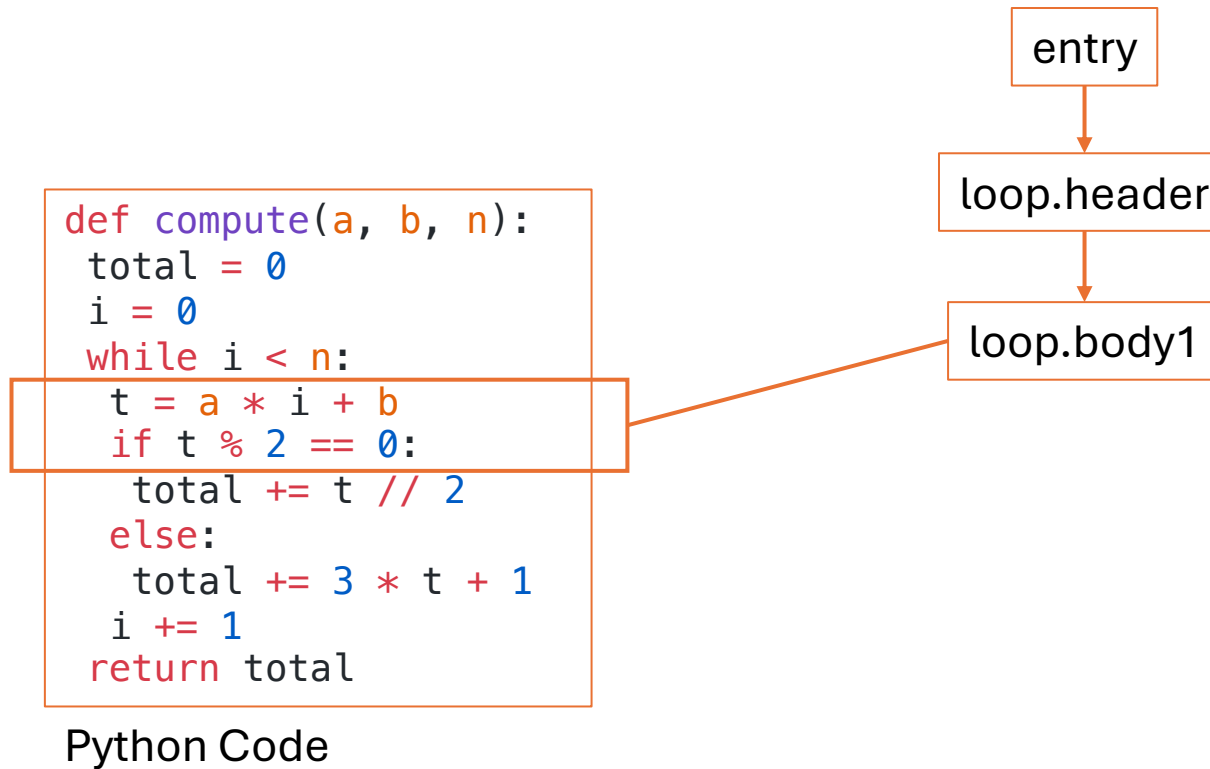
Control Flow Graph



Python Code

Control Flow Graph

Control Flow Graph

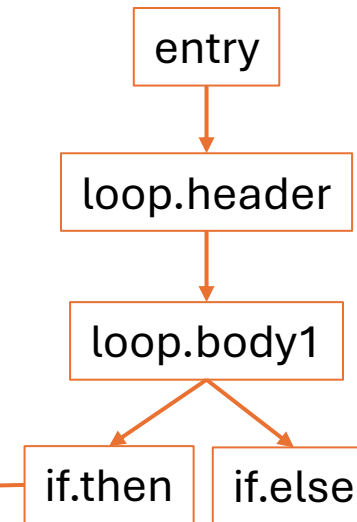


Control Flow Graph

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
            i += 1  
    return total
```

Python Code

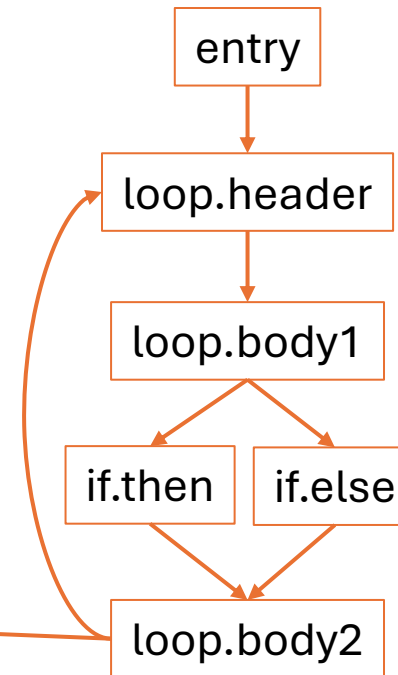


Control Flow Graph

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

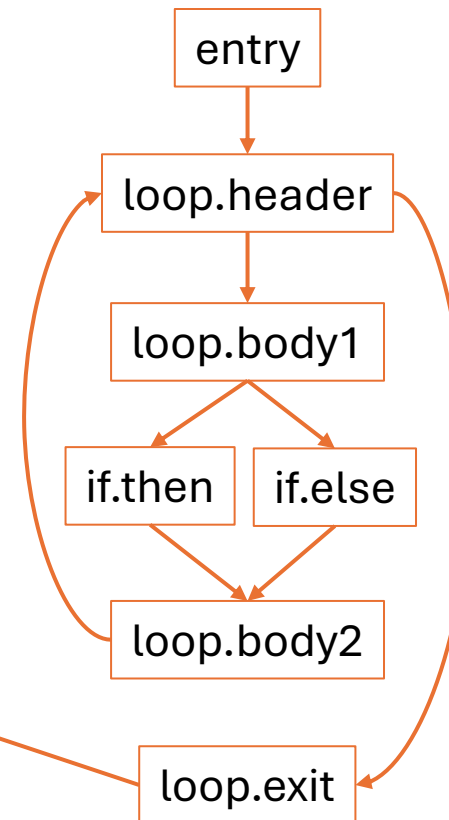


Control Flow Graph

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

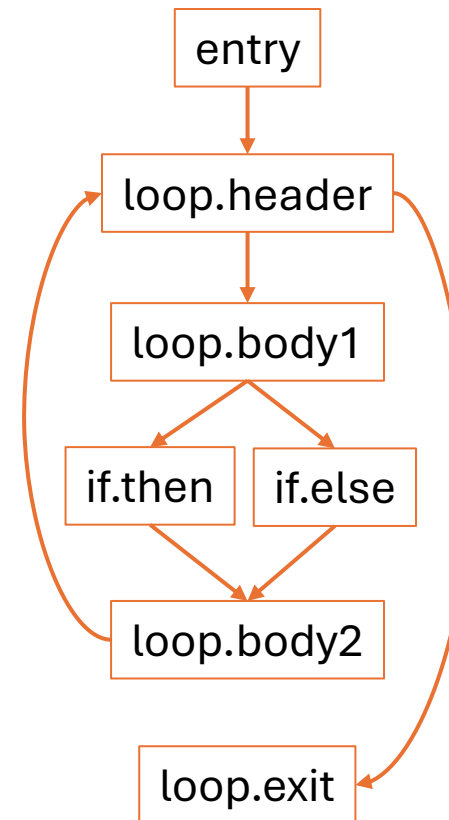


Control Flow Graph

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code



Control Flow Graph

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

```
int compute(int a, int b, int n) {  
    int total = 0;  
    int i = 0;  
    while (i < n) {  
        int t = a * i + b;  
        if ((t % 2) == 0) {  
            total += t / 2;  
        } else {  
            total += 3 * t + 1;  
        }  
        i += 1;  
    }  
    return total;  
}
```

C Code

Control Flow Graph

```
def compute(a, b, n):  
    total = 0  
    i = 0  
    while i < n:  
        t = a * i + b  
        if t % 2 == 0:  
            total += t // 2  
        else:  
            total += 3 * t + 1  
        i += 1  
    return total
```

Python Code

```
int compute(int a, int b, int n) {  
    int total = 0;  
    int i = 0;  
    while (i < n) {  
        int t = a * i + b;  
        if ((t % 2) == 0) {  
            total += t / 2;  
        } else {  
            total += 3 * t + 1;  
        }  
        i += 1;  
    }  
    return total;  
}
```

C Code

```
define i64 @compute(i64 %a, i64 %b, i64 %n) {  
entry:  
    %total = alloca i64, align 8  
    %i = alloca i64, align 8  
    %t = alloca i64, align 8  
    store i64 0, i64* %total  
    store i64 0, i64* %i  
    br label %loop.header  
  
loop.header: ; while (i < n)?  
    %i.val = load i64, i64* %i  
    %cmp = icmp slt i64 %i.val, %n  
    br i1 %cmp, label %loop.body, label %loop.exit  
  
loop.body:  
    ; t = a*i + b  
    %i.val2 = load i64, i64* %i  
    %mul = mul i64 %a, %i.val2  
    %add = add i64 %mul, %b  
    store i64 %add, i64* %t  
  
    ; if ((t % 2) == 0) ...  
    %t.val = load i64, i64* %t  
    %rem = srem i64 %t.val, 2  
    %isEven = icmp eq i64 %rem, 0  
    br i1 %isEven, label %then.block, label %else.block  
  
then.block:  
    ; total += t / 2  
    %t.even = load i64, i64* %t  
    %half = sdiv i64 %t.even, 2  
    %tot0 = load i64, i64* %total  
    %tot1 = add i64 %tot0, %half  
    store i64 %tot1, i64* %total  
    br label %cont.block  
  
else.block:  
    ; total += 3*t + 1  
    %t.odd = load i64, i64* %t  
    %triple = mul i64 %t.odd, 3  
    %plus1 = add i64 %triple, 1  
    %tot2 = load i64, i64* %total  
    %tot3 = add i64 %tot2, %plus1  
    store i64 %tot3, i64* %total  
    br label %cont.block  
  
cont.block:  
    ; i += 1  
    %i.cur = load i64, i64* %i  
    %i.next = add i64 %i.cur, 1  
    store i64 %i.next, i64* %i  
    br label %loop.header  
  
loop.exit:  
    %tot.final = load i64, i64* %total  
    ret i64 %tot.final  
}
```

LLVM Code (Compiled)

Control Flow Graph

```
define i64 @compute(i64 %a, i64 %b, i64 %n) {
entry:
%total = alloca i64, align 8
%i = alloca i64, align 8
%t = alloca i64, align 8
store i64 0, i64* %total
store i64 0, i64* %i
br label %loop.header

loop.header: ; while (i < n)?
%i.val = load i64, i64* %i
%cmp = icmp slt i64 %i.val, %n
br i1 %cmp, label %loop.body, label %loop.exit

loop.body:
; t = a*i + b
%i.val2 = load i64, i64* %i
%mul = mul i64 %a, %i.val2
%add = add i64 %mul, %b
store i64 %add, i64* %t

; if ((t % 2) == 0) ...
%t.val = load i64, i64* %t
%rem = srem i64 %t.val, 2
%isEven = icmp eq i64 %rem, 0
br i1 %isEven, label %then.block, label %else.block

then.block:
; total += t / 2
%t.even = load i64, i64* %t
%half = sdiv i64 %t.even, 2
%tot0 = load i64, i64* %total
%tot1 = add i64 %tot0, %half
store i64 %tot1, i64* %total
br label %cont.block

else.block:
; total += 3*t + 1
%t.odd = load i64, i64* %t
%triple = mul i64 %t.odd, 3
%plus1 = add i64 %triple, 1
%tot2 = load i64, i64* %total
%tot3 = add i64 %tot2, %plus1
store i64 %tot3, i64* %total
br label %cont.block

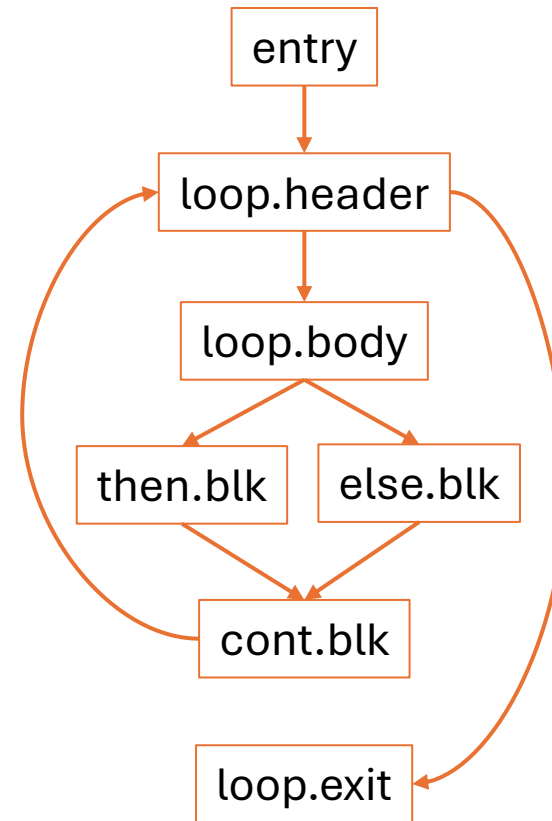
cont.block:
; i += 1
%i.cur = load i64, i64* %i
%i.next = add i64 %i.cur, 1
store i64 %i.next, i64* %i
br label %loop.header

loop.exit:
%tot.final = load i64, i64* %total
ret i64 %tot.final
}
```

LLVM Code (Compiled)

Control Flow Graph

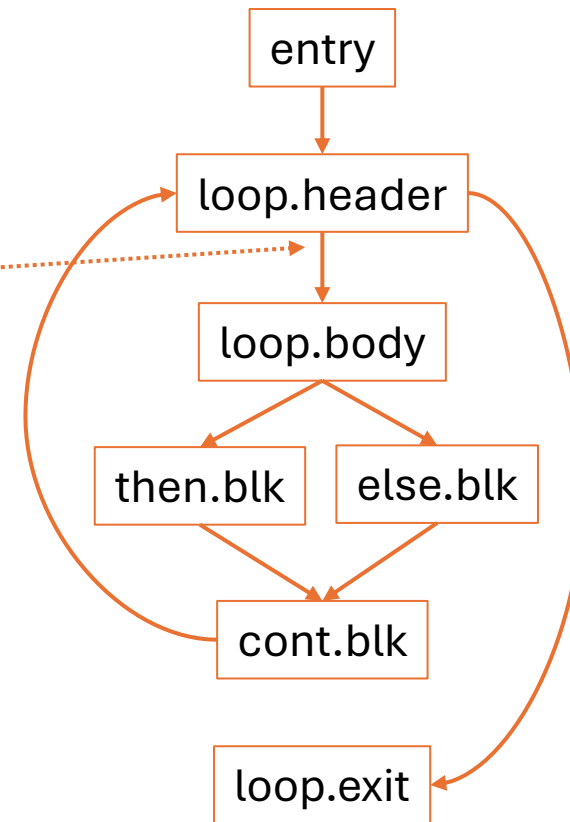
```
loop.header: ; while (i < n)?  
%i.val = load i64, i64* %i  
%cmp = icmp slt i64 %i.val, %n  
br i1 %cmp, label %loop.body, label %loop.exit  
  
loop.body:  
; t = a*i + b  
%i.val2 = load i64, i64* %i  
%mul = mul i64 %a, %i.val2  
%add = add i64 %mul, %b  
store i64 %add, i64* %t  
  
; if ((t % 2) == 0) ...  
%t.val = load i64, i64* %t  
%rem = srem i64 %t.val, 2  
%isEven = icmp eq i64 %rem, 0  
br i1 %isEven, label %then.block, label %else.block
```



Control Flow Graph

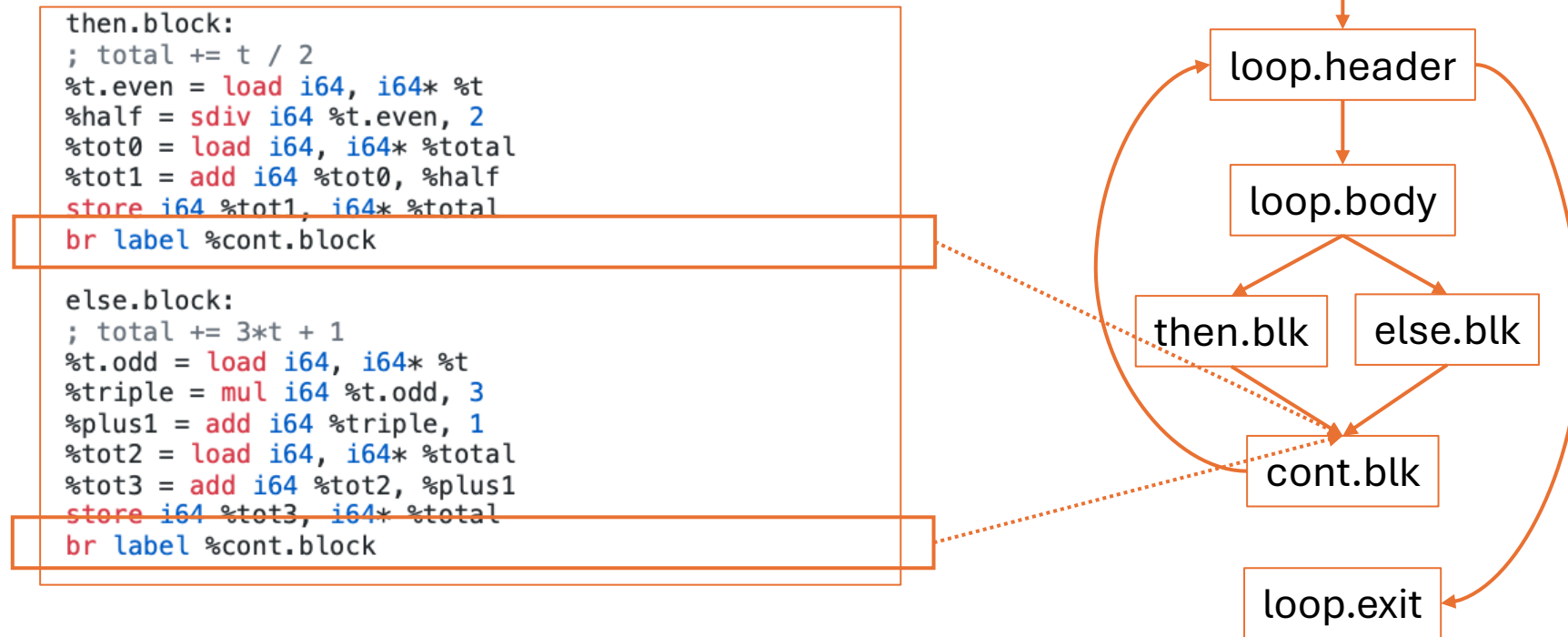
Control Flow Graph

```
loop.header: ; while (i < n)?  
%i.val = load i64, i64* %i  
%cmp = icmp slt i64 %i.val, %n  
br i1 %cmp, label %loop.body, label %loop.exit  
  
loop.body:  
; t = a*i + b  
%i.val2 = load i64, i64* %i  
%mul = mul i64 %a, %i.val2  
%add = add i64 %mul, %b  
store i64 %add, i64* %t  
  
; if ((t % 2) == 0) ...  
%t.val = load i64, i64* %t  
%rem = srem i64 %t.val, 2  
%isEven = icmp eq i64 %rem, 0  
br i1 %isEven, label %then.block, label %else.block
```



Control Flow Graph

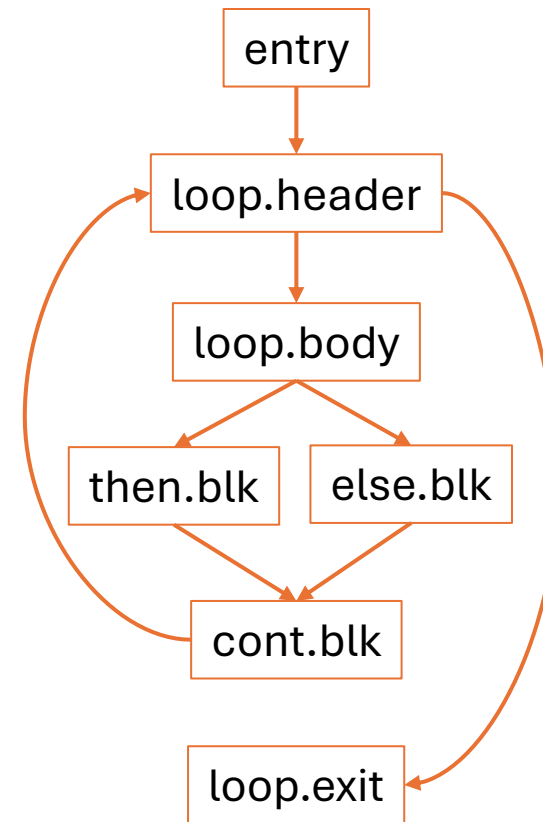
Control Flow Graph



Control Flow Graph

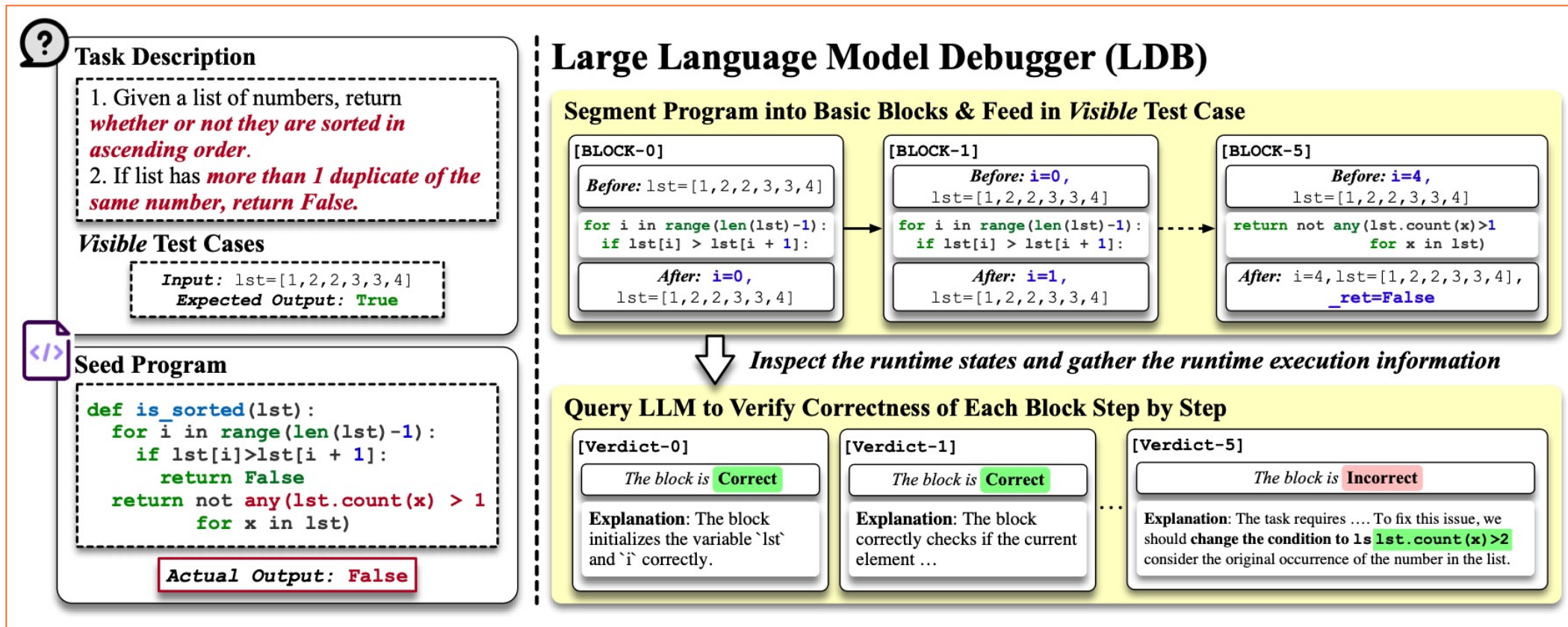
Control Flow Graph: Key Takeaway

- Representing program differently helps analysis of program behavior
- Is a more concise representation because multiple instructions are grouped into a **basic block**
- Control flow graph identifies how program is executed, helping to perform **error localization**

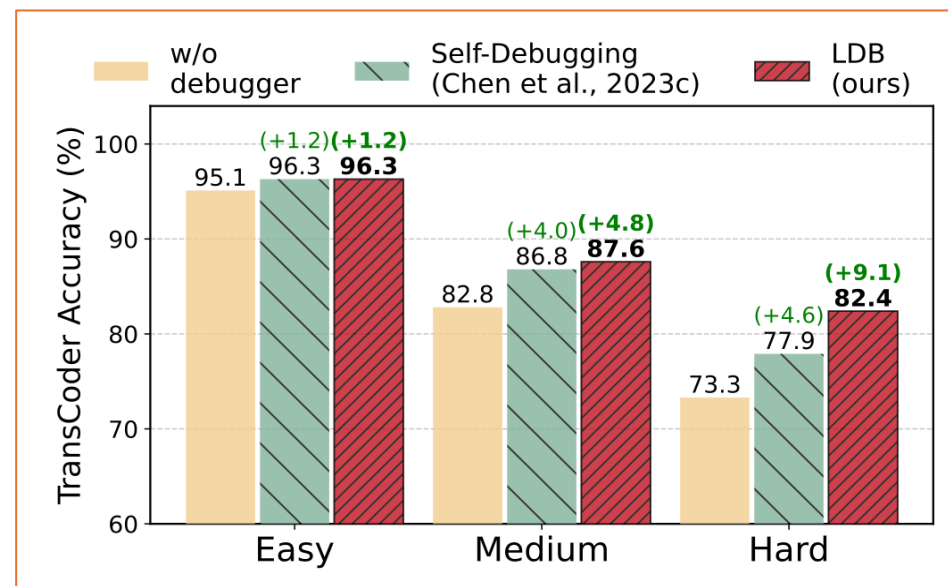
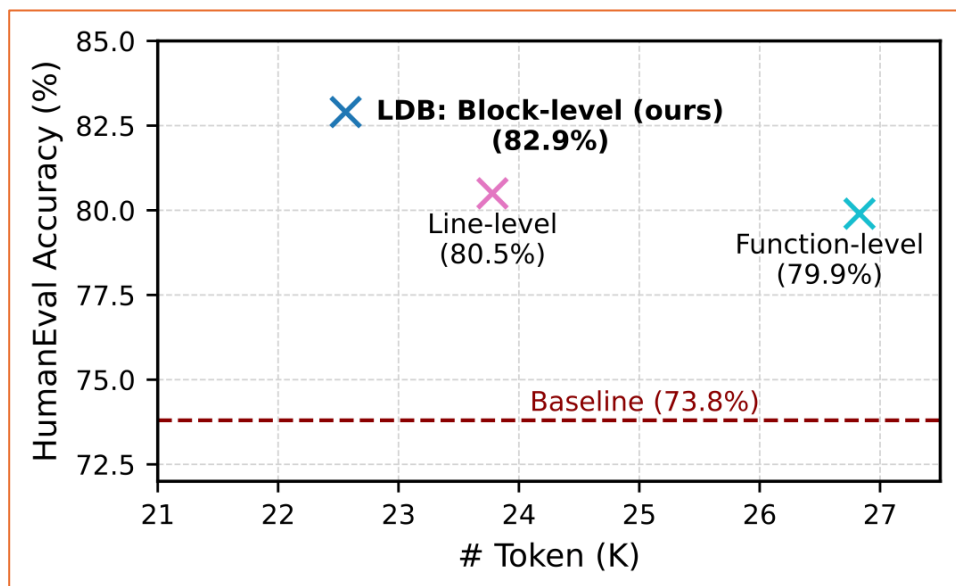


Control Flow Graph

Debugger for Feedback



Debugger for Feedback



More Unit Tests

- What if the provided unit tests are insufficient?
 - Does not cover edge cases
 - Does not guarantee correct solution

```
def is_palindrome(s: str) -> bool:
    """
    return True if s reads the same forwards and
    backwards. Ignore case and whitespace or
    punctuation.
    """
    return s == s[::-1] # Syntheized test case

class TestPalindrome(unittest.TestCase):
    def test_racecar_capitalized(self):
        # case-insensitive.
        self.assertTrue(is_palindrome("Racecar"))
```

Toward Trustworthy Neural Program Synthesis

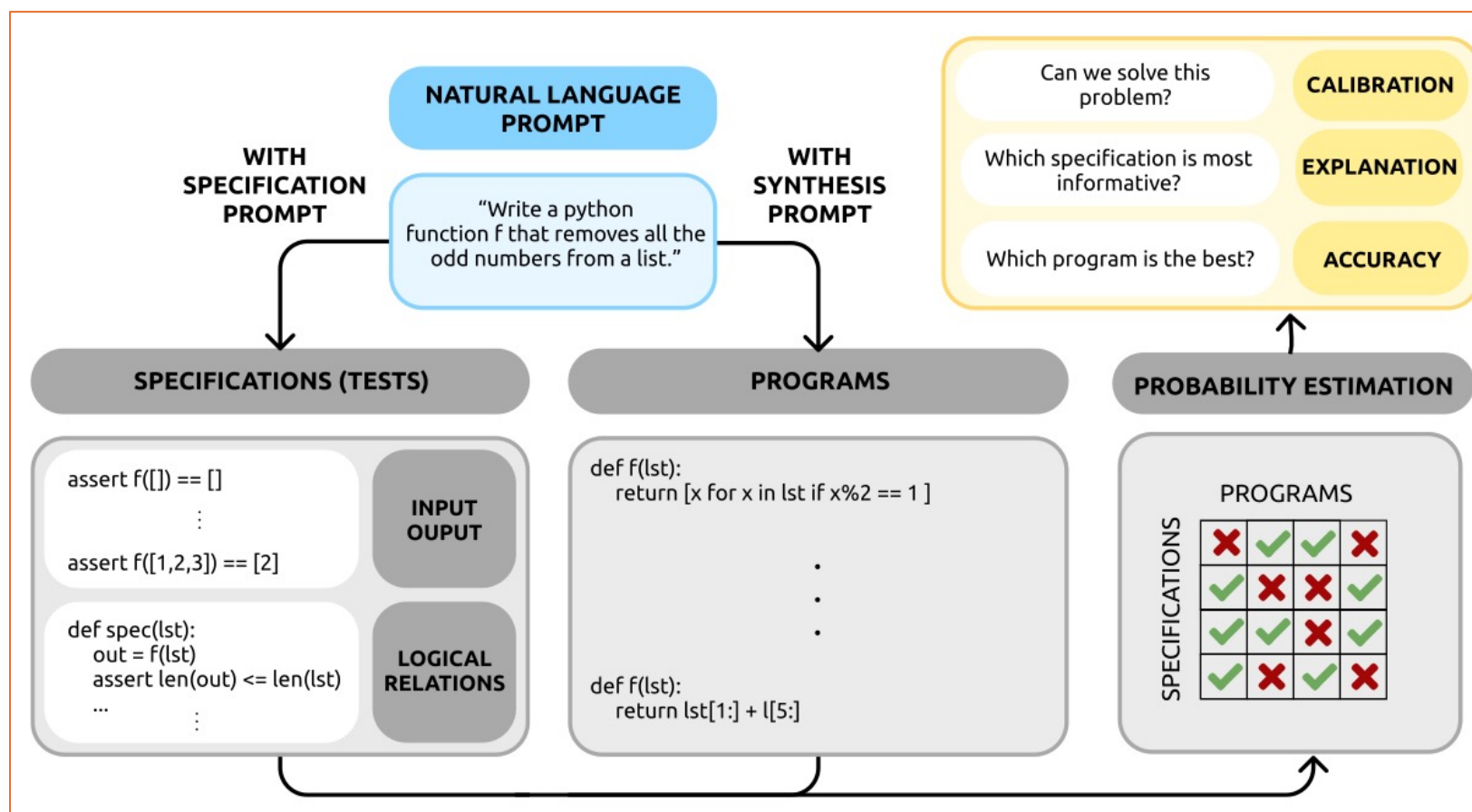
Wen-Ding Li^{*1}, Darren Key^{*1}, Kevin Ellis¹

¹Department of Computer Science, Cornell University, USA
wl678@cornell.edu, dyk34@cornell.edu, kellis@cornell.edu

Toward Trustworthy Neural Program Synthesis

Wen-Ding Li^{*1}, Darren Key^{*1}, Kevin Ellis¹

¹Department of Computer Science, Cornell University, USA
wl678@cornell.edu, dyk34@cornell.edu, kellis@cornell.edu



Toward Trustworthy Neural Program Synthesis

Wen-Ding Li^{*1}, Darren Key^{*1}, Kevin Ellis¹

¹Department of Computer Science, Cornell University, USA
wl678@cornell.edu, dyk34@cornell.edu, kellis@cornell.edu

Generating Program

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    return list(map(lambda x, y: x-y, nums1, nums2))
```

Generating Input-Output Specification

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    pass # To-do: implement

# Check if sub_list works
assert sub_list([2, 3, 1], [1, 1, 1]) == [1, 2, 0])
```

Toward Trustworthy Neural Program Synthesis

Generating Program

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    return list(map(lambda x, y: x-y, nums1, nums2))
```

Generating Input-Output Specification

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    pass # To-do: implement

# Check if sub_list works
assert sub_list([2, 3, 1], [1, 1, 1]) == [1, 2, 0])
```

Generating Logical-Relation Specification

```
[...FEWSHOT EXAMPLES...]
# Problem 3
# Write a function to subtract two lists element-wise.
def sub_list(nums1,nums2):
    pass # To-do: implement
# Test 3
def test_sub_list(nums1 : list, nums2 : list):
    """
    Given two lists `nums1` and `nums2`, test whether function `sub\_list` is implemented correctly.
    """
    output_list = sub_list(nums1, nums2)
    # check if the length of the output list is the same as the lengths of the input lists
    assert len(output_list) == len(nums1) == len(nums2)
    # check if the output list has the expected elements
    for i in range(len(output\_list)):
        assert output_list[i] == nums1[i] - nums2[i]

# run the testing function `test_sub_list` on a new testcase
test_sub_list([1, 2, 3, 4], [10, 9, 8, 7])
```

Property Based Testing

Verification of Program with a Specification

{Pre-condition} Program {Post-condition}

$$\{P\} c \{Q\}$$

Program c : a program state transformer

Pre- and Post-condition P, Q : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

Property Based Testing

- Issues with verification with functional specifications:
 - Very hard to specify a **complete** functional specification
 - Pre- and post-conditions sometimes require **special constructs**
 - Requires a **formal theorem prover** / solver to verify

Verification of Program with a Specification

{Pre-condition} Program **{Post-condition}**

{P} *c* **{Q}**

Program *c*: a program state transformer

Pre- and Post-condition *P*, *Q*: a Boolean function over program states

$\forall x, P(x) \Rightarrow Q(c(x))$

Property Based Testing

- Issues with verification with functional specifications:
 - Very hard to specify a **complete** functional specification
 - Pre- and post-conditions sometimes require **special constructs**
 - Requires a **formal theorem prover** / solver to verify
- Solution: **Property Based Testing!**

Verification of Program with a Specification

{Pre-condition} Program **{Post-condition}**

{P} *c* **{Q}**

Program *c*: a program state transformer

Pre- and Post-condition *P*, *Q*: a Boolean function over program states

$\forall x, P(x) \Rightarrow Q(c(x))$

Property Based Testing

- Issues with verification with functional specifications:
 - Very hard to specify a **complete** functional specification
 - Pre- and post-conditions sometimes require **special constructs**
 - Requires a **formal theorem prover / solver** to verify
- Solution: **Property Based Testing!**

Property-Based Testing in Practice

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Joseph W. Cutler
University of Pennsylvania
Philadelphia, PA, USA
jwc@seas.upenn.edu

Daniel Dickstein
Jane Street
New York, NY, USA
ddickstein@janestreet.com

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

Program c : a program state transformer

Pre- and Post-condition P, Q : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
 $P$      $x$ : Int  
  
    int abs(int  $x$ ) {  
        int  $y$ ;  
        if ( $x$  >= 0) // B1  
 $C$          $y$  =  $x$ ; // B2  
        else  
             $y$  = - $x$ ; // B3  
        return  $y$ ; // B4  
    }
```

```
 $Q$     ( $y$  = - $x$   $\vee$   $y$  =  $x$ )  $\wedge$   $y$   $\geq$  0
```

Program c : a program state transformer

Pre- and Post-condition P, Q : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

P $x: \text{Int}$

```
int abs(int x) {  
    int y;  
    if (x >= 0) // B1  
        y = x; // B2  
    else  
        y = -x; // B3  
    return y; // B4  
}
```

c

Q $(y = -x \vee y = x) \wedge y \geq 0$

```
bool abs_property(int x, int y) {  
    return (x == y || x == -y) &&  
           y >= 0;  
}
```

Property of abs encoded as Boolean function (predicate)

Program c : a program state transformer

Pre- and Post-condition P, Q : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

P $x: \text{Int}$

```
int abs(int x) {  
    int y;  
    if (x >= 0) // B1  
        y = x; // B2  
    else  
        y = -x; // B3  
    return y; // B4  
}
```

C

Q $(y = -x \vee y = x) \wedge y \geq 0$

```
bool abs_property(int x, int y) {  
    return (x == y || x == -y) &&  
           y >= 0;  
}
```

Property of abs encoded as Boolean function (predicate)

```
for (int i = 0; i < 1000; i++) {  
    int x = random_integer();  
    int y = abs(x);  
    assert(abs_property(x, y));  
}
```

Perform random testing with the property

Toward Trustworthy Neural Program Synthesis

Generating Program

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    return list(map(lambda x, y: x-y, nums1, nums2))
```

Generating Input-Output Specification

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-wise.
    """
    pass # To-do: implement

# Check if sub_list works
assert sub_list([2, 3, 1], [1, 1, 1]) == [1, 2, 0])
```

Generating Logical-Relation Specification

```
[...FEWSHOT EXAMPLES...]
# Problem 3
# Write a function to subtract two lists element-wise.
def sub_list(nums1,nums2):
    pass # To-do: implement
# Test 3
def test_sub_list(nums1 : list, nums2 : list):
    """
    Given two lists `nums1` and `nums2`, test whether function `sub\_list` is implemented correctly.
    """
    output_list = sub_list(nums1, nums2)
    # check if the length of the output list is the same as the lengths of the input lists
    assert len(output_list) == len(nums1) == len(nums2)
    # check if the output list has the expected elements
    for i in range(len(output\_list)):
        assert output_list[i] == nums1[i] - nums2[i]

# run the testing function `test_sub_list` on a new testcase
test_sub_list([1, 2, 3, 4], [10, 9, 8, 7])
```

Verification as Feedback

- Issues with verification with functional specifications:
 - Very hard to specify a **complete** functional specification
 - Pre- and post-conditions sometimes require **special constructs**
 - Requires a **formal theorem prover** / solver to verify

Verification of Program with a Specification

{Pre-condition} Program {Post-condition}

$$\{P\} c \{Q\}$$

Program c : a program state transformer

Pre- and Post-condition P, Q : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

Verified Code Transpilation with LLMs

Sahil Bhatia¹ Jie Qiu Niranjan Hasabnis^{2*} Sanjit A. Seshia¹ Alvin Cheung¹

¹UC Berkeley ²Code Metal

{sahilbhatia, jieq, sseshia, akcheung}@berkeley.edu

niranjan@codemetal.ai

Verified Code Transpilation with LLMs

Sahil Bhatia¹ Jie Qiu Niranjan Hasabnis^{2*} Sanjit A. Seshia¹ Alvin Cheung¹
¹UC Berkeley ²Code Metal
{sahilbhatia, jieq, sseshia, akcheung}@berkeley.edu
niranjan@codemetal.ai

Code Transpilation

Language X $\longrightarrow$ Language Y

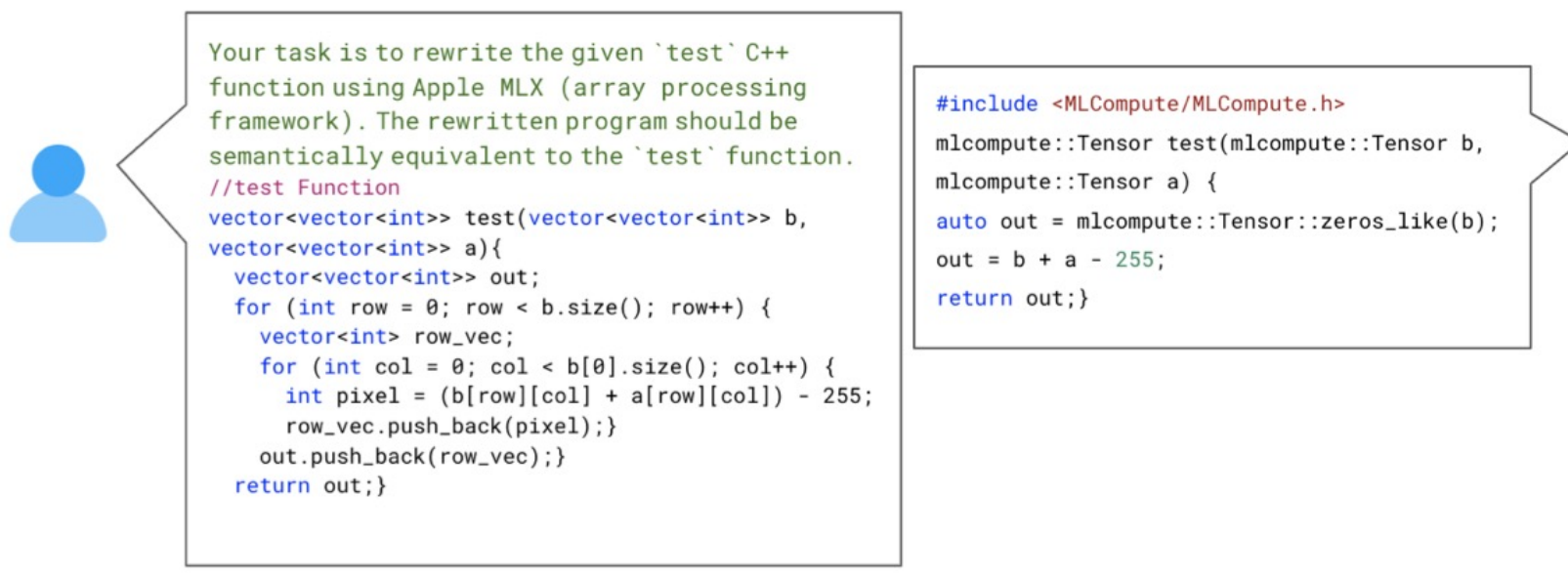
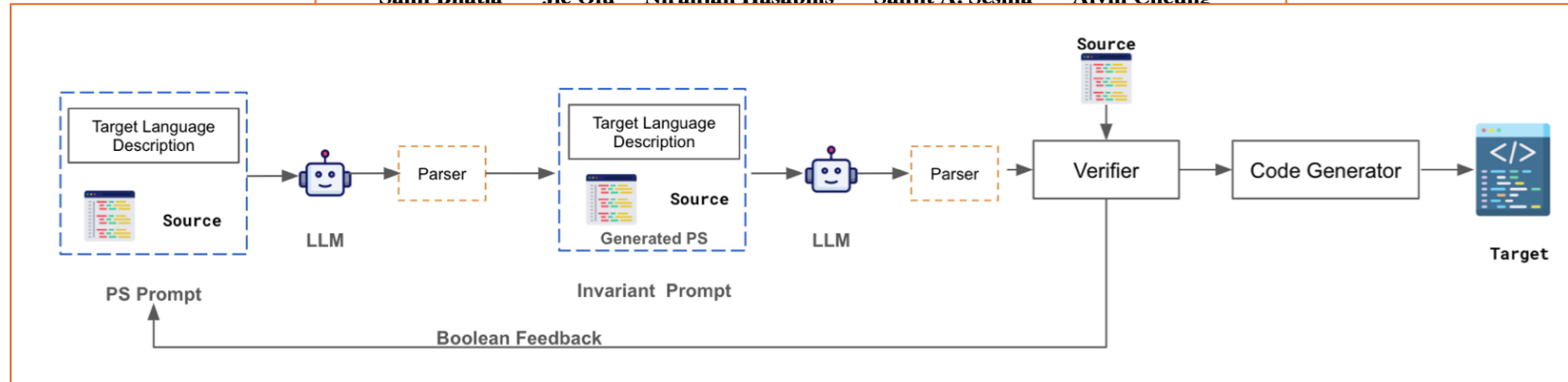
Given program $x \in X$ $\longrightarrow$ Synthesize $y \in Y$

Specification

$$\forall i, \text{eval}_X(x, i) = \text{eval}_Y(y, i)$$

Verified Code Transpilation with LLMs

Sahil Bhatia¹ Jie Oiu Niranian Hasabnis^{2*} Saniit A. Seshia¹ Alvin Cheung¹



Summary of Iterative Refinement Feedback

- Compiler messages
- Linter messages
- Unit test report
 - Generating traces from unit test using debuggers
 - Simplifying traces via control flow graph
 - Generating more test cases
- Property based testing
- Verification result

IS SELF-REPAIR A SILVER BULLET FOR CODE GENERATION?

**Theo X. Olausson^{1, †}, Jeevana Priya Inala², Chenglong Wang²,
Jianfeng Gao², Armando Solar-Lezama¹**

¹MIT CSAIL ²Microsoft Research

IS SELF-REPAIR A SILVER BULLET FOR CODE GENERATION?

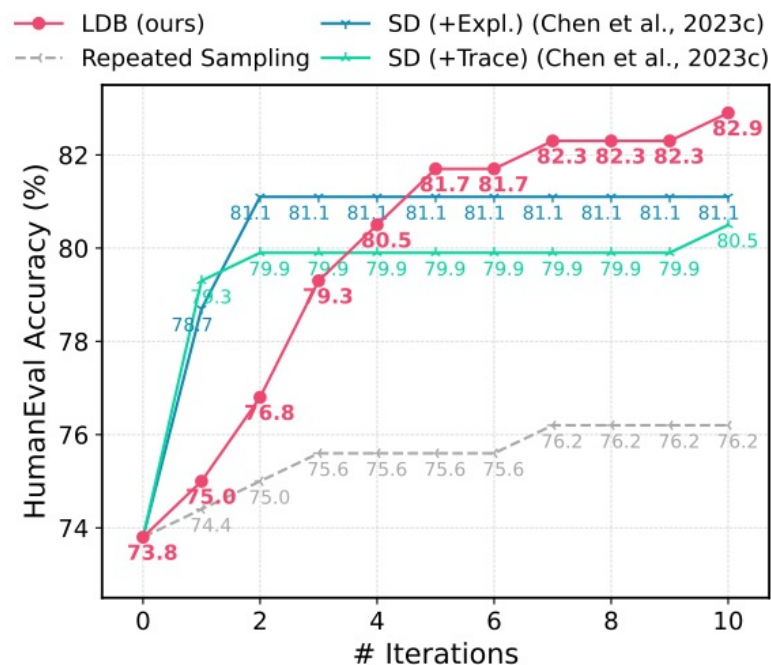
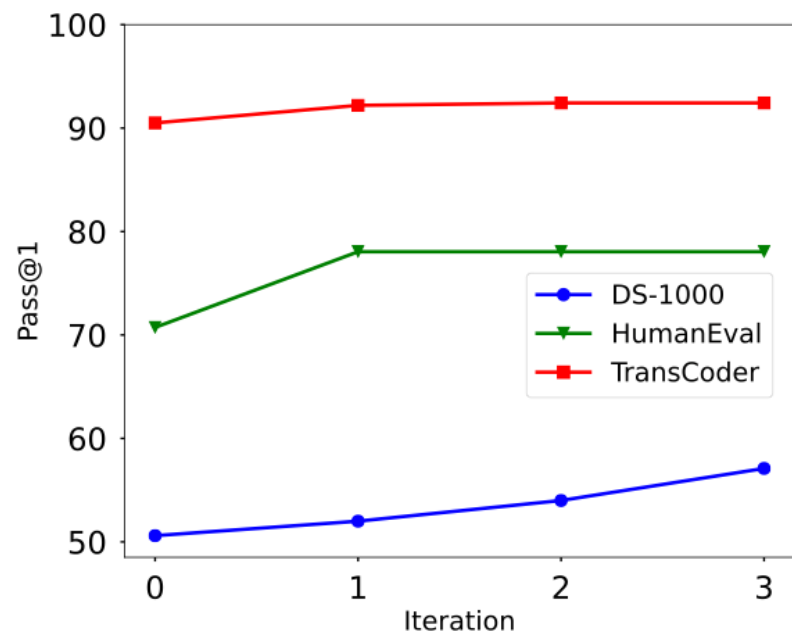
**Theo X. Olausson^{1, †}, Jeevana Priya Inala², Chenglong Wang²,
Jianfeng Gao², Armando Solar-Lezama¹**

¹MIT CSAIL ²Microsoft Research

- Self-repair is not a silver bullet: when the cost of repair is taken into account, we find several instances in which pass rates are higher or equally high with i.i.d. sampling (without repair), especially when the budget is small. We conjecture that this is because program generation and repair rates trend together, and many subtle factors influence which one will overpower the other for a given task (see Appendix [C](#)).

Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step

Li Zhong Zilong Wang[†] Jingbo Shang[†]
University of California, San Diego
{lizhong, zlwang, jshang}@ucsd.edu



SELF EVOLVE: A Code Evolution Framework via Large Language Models

Shuyang Jiang¹, Yuhao Wang¹, Yu Wang^{1,2*}

¹Shanghai Jiao Tong University

²Shanghai AI Laboratory

{jiangshuyang, colane, yuwangs@sjtu}@sjtu.edu.cn

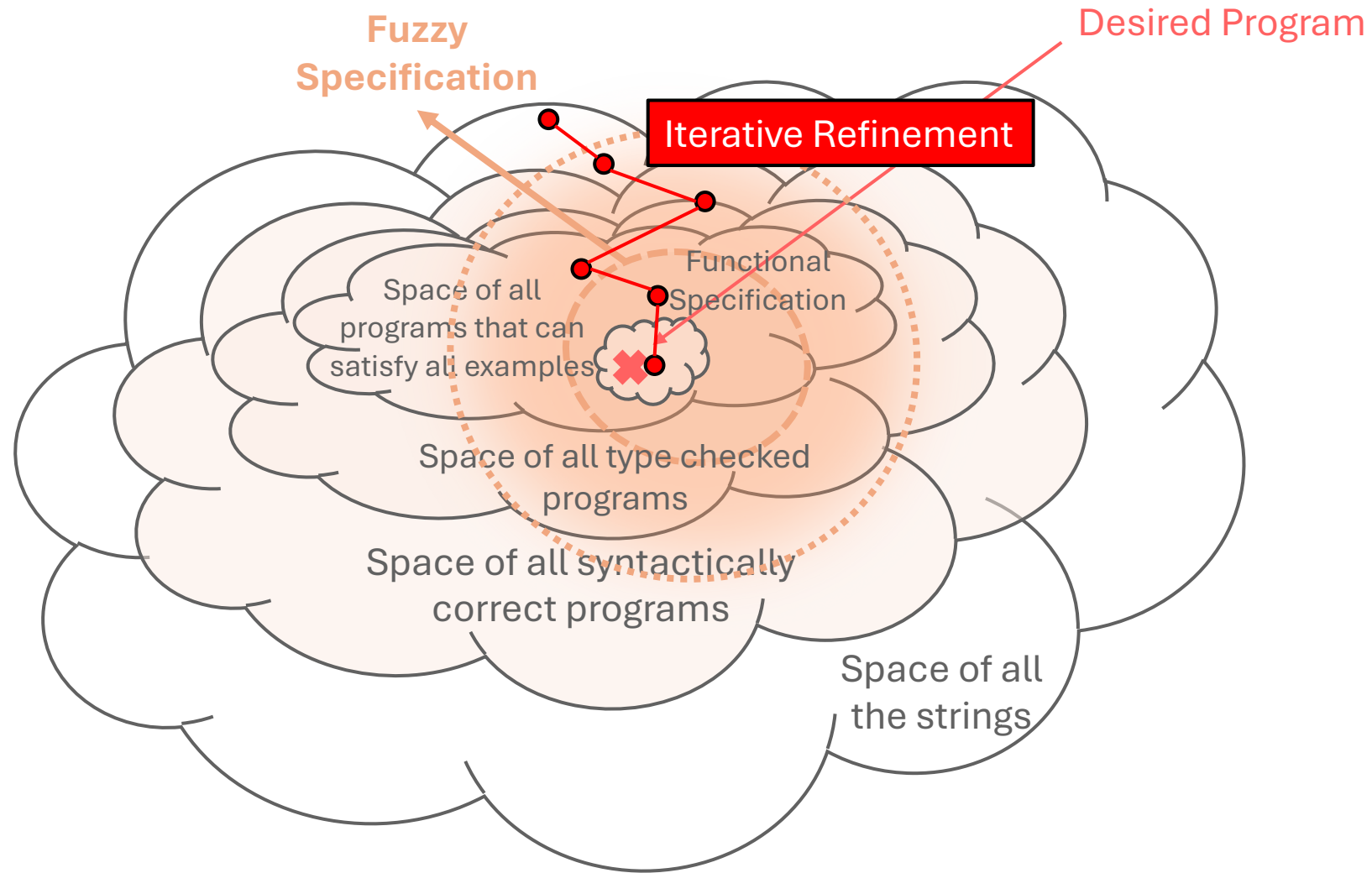
IS SELF-REPAIR A SILVER BULLET FOR CODE GENERATION?

- Self-repair is not a silver bullet: when the cost of repair is taken into account, we find several instances in which pass rates are higher or equally high with i.i.d. sampling (without repair), especially when the budget is small. We conjecture that this is because program generation and repair rates trend together, and many subtle factors influence which one will overpower the other for a given task (see Appendix C).
- Artificially boosting the quality of the feedback significantly improves the efficacy of self-repair. We replace Code Llama’s feedback with that produced by GPT-3.5 or GPT-4, and GPT-3.5’s feedback with that of GPT-4; in every case, the boosted configuration beats both the corresponding i.i.d. baseline and the corresponding self-repair configuration at all budgets. Furthermore, replacing GPT-4’s own explanations with those of a human programmer improves repair significantly, increasing the fraction of repaired programs which pass the tests by a factor of $1.58\times$ (from 33.3% to 52.6%).

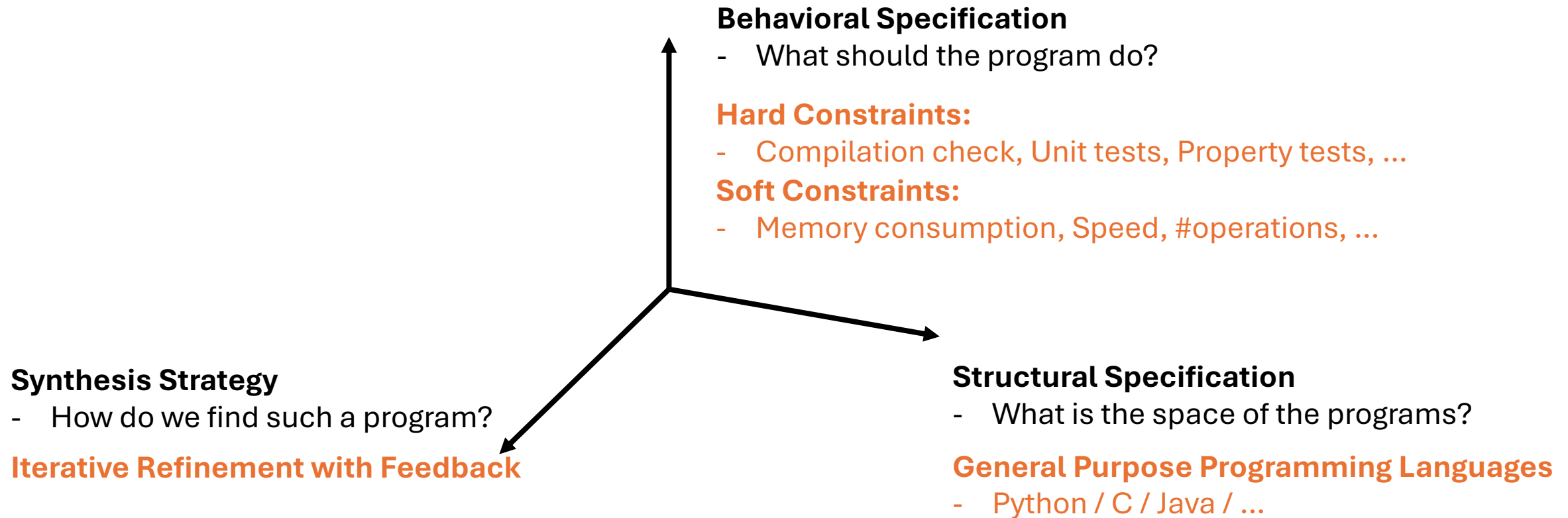
Table 1: Success rate of repair with GPT-4’s explanations vs. with those of our human participants.

Difficulty	Introductory	Interview	Competition	Overall
GPT-4 Feedback	42.64%	19.33%	3.67%	33.30%
Human Feedback	62.21%	45.67%	14.67%	52.60%

High Level Picture



Dimensions in Program Synthesis



Week 4

- Assignment 1
 - <https://github.com/machine-programming/assignment-1>
 - Due Today!
- Assignment 2
 - <https://github.com/machine-programming/assignment-2>
 - Due ~two weeks from now (Oct 2)
 - Evaluating Coding LLMs