# Learning to Synthesize

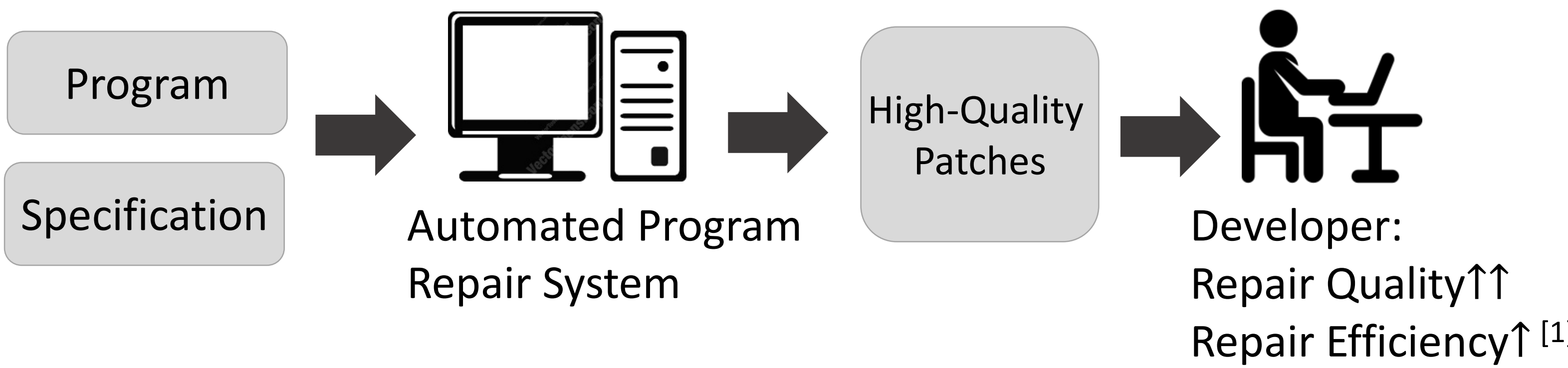
Yingfei Xiong
Peking University
IBF 2019

# Bug Fixing Costs a Lot

- Developers spend 50% of their time debugging[1]
- The development team often does not have enough resource for bug-fixing [2]
- Software is often released with known bugs [3]

[1] Britton et al. Quantify the time and cost saved using reversible debuggers. Cambridge report, 2013
[2] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” eXchange, 2005, pp. 35–39
[3] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in PLDI, 2003, pp. 141–154

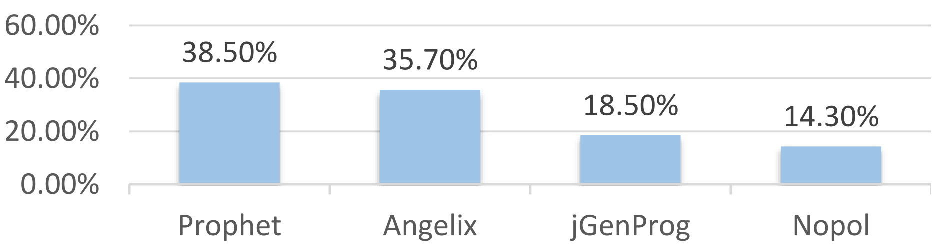
# Automated Program Repair



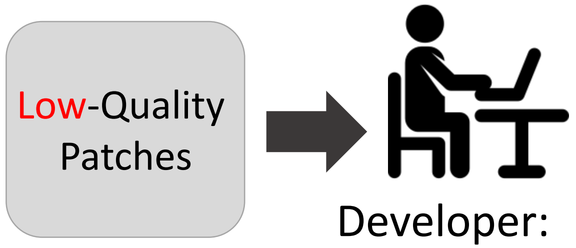

[1] Yida Tao, Jindae Kim, Sunghun Kim, Chang Xu: Automatically generated patches as debugging aids: a human study. SIGSOFT FSE 2014: 64-74

# Weak Specification Problem

- Programs usually have only weak specification such as tests.
- Early systems aim to meet the specification, often producing low-quality patches.




Precisions of some popular systems
(before 2016)




Developer:
Repair Quality↓↓
Repair Efficiency↓[1]

[1] Yida Tao, Jindae Kim, Sunghun Kim, Chang Xu: Automatically generated patches as debugging aids: a human study. SIGSOFT FSE 2014: 64-74

# How to deal with the weak specification?

- Find the most-likely patch under the current context
- Precisions of Recent tools:
  - ACS [1] +Patch Filtering [2] : 85%
  - ConCap [3] : 84%
- This talk:
  - A generalization of this weak specification problem
  - A general framework to address this problem

[1] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, Lu Zhang. Precise Condition Synthesis for Program Repair. ICSE'17.
[2] Yingfei Xiong, Xinyuan Liu#, Muhan Zeng#, Lu Zhang, Gang Huang. Identifying Patch Correctness in Test-Based Program Repair. ICSE'18.
[3] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, Shing-Chi Cheung: Context-aware patch generation for better automated program repair. ICSE'18.

# Program Estimation

- We aim to find the program that are most-likely to be written under the current context.

```
public static long factorial(final int n){
    if( ... ){   }
}
```

```
...
Math.abs(n) < 1
n == Integer.Max_VALUE
n < 19
n < 21
...
```

- We define this problem as **program estimation**:
  - Given a context $c$, a (weak) specification E, and a space of programs S,
    find program $s = \text{argmax}_{s \in S \wedge E(s)} P(s \mid c)$?
- A sub-problem of program synthesis

# Application: Test-based Program Repair

- Context = buggy program & at least one failed test

Passing Test

Failed Test

Buggy code

```
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```

# Application: Code Completion

- Context = partial code

```
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

# Application: Program by Examples

- Context = input/output examples

| Input | Output |
|---|---|
| 0d 5h 26m | 5:00 |
| 0d 4h 57m | 4:30 |
| 0d 4h 27m | 4:00 |
| 0d 3h 57m | 3:30 |

# Application: Code Generation from Natural Language

- Context = natural language description

```
/**
 * Internal helper method for natural logarithm function.
 * @param x original argument of the natural logarithm function
 * @param hiPrec extra bits of precision on output (To Be Confirmed)
 * @return log(x)
 */
```

# Application: Test Generation

- Context = program under test
- Probability = bug-detection capability

```
public int add(int a, int b) {
  …
}
```

Context

```
public void testAdd() {
 …
}
```

Program to be generated

# Challenges

- How to estimate the probability $P(Prog \mid Context)$?

- How to find program $s$ such that $E(s)$ and $P( s \mid context )$ is the largest?

# Learning to synthesis (L2S)

- A general framework to address program estimation

- Combining four tools
  - **Rewriting rules**: defining a search problem
  - **Constraint solving**: pruning off invalid choices in each step
  - **Machine-learned models**: estimating the probabilities of choices in each step
  - **Search algorithms**: solving the search problem

# Example: Condition Completion

- Given a program without a conditional expression, completing the condition

```
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

E → E ">12"
  | E ">0"
  | E "+" E
  | *"hours"*
  | *"value"*
  | …

Space of Conditions

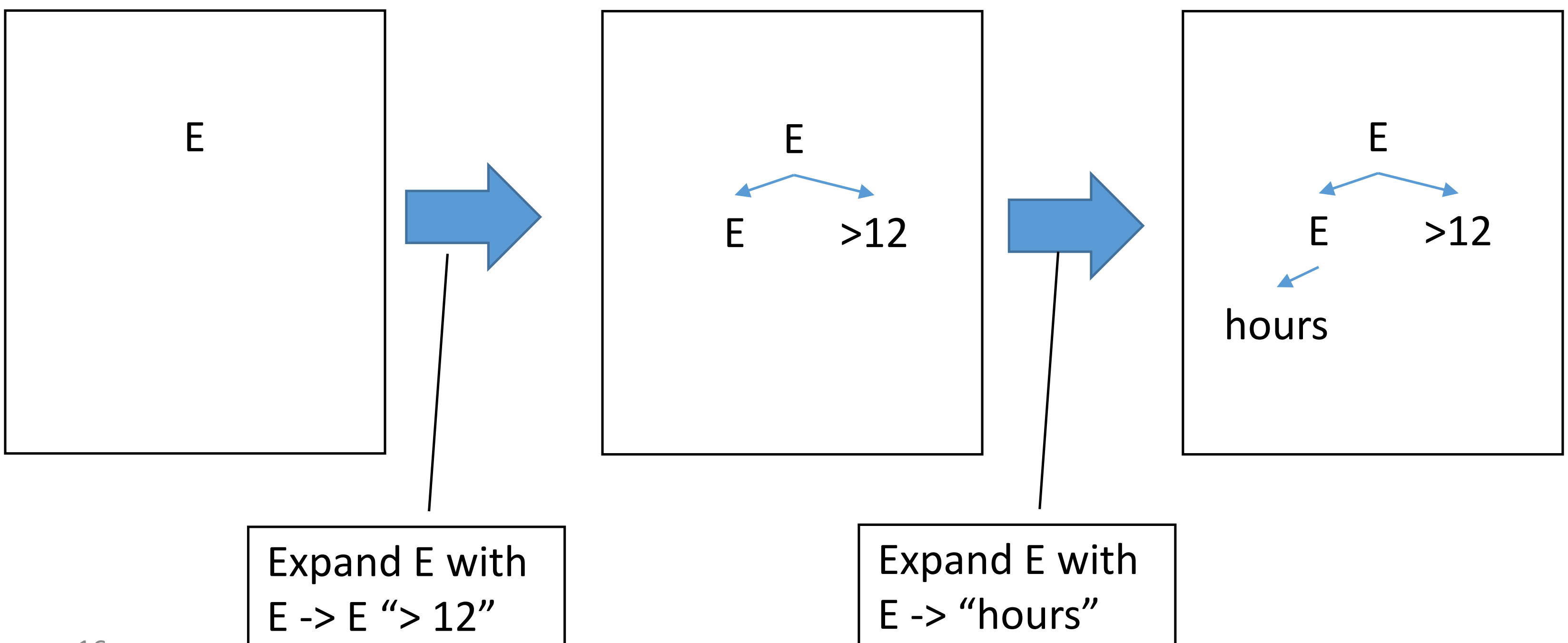
- Useful in program repair
  - Many bugs are caused by incorrect conditions
  - Existing work could localize the faulty condition
  - Can we generate a correct condition to replace the incorrect one?

# Challenge 1: Estimating the Probability

- Idea: Using machine learning
  - To train over a set of programs and their contexts
- Problem: machine learning usually works for classification problems
  - where the number of classes are usually small
- Idea: turn the generation problem into a set of classification problem along the grammar

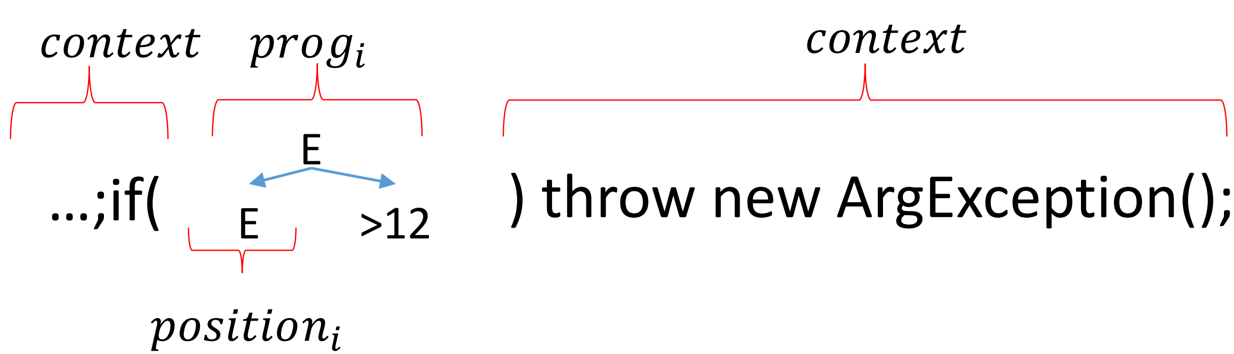
# Decomposing Generation

- In each step, we estimate the probabilities of the rules to expand the left-most non-terminal
  - A classification problem



# Probability of the program

- $P(prog \mid context) = \prod_i P(rule_i \mid context, prog_i, position_i)$
  - $context$: The context of the program
  - $prog_i$: The AST generated at the ith step
  - $position_i$: The non-terminal to be expanded at the ith step
  - rule: the chosen rule at the ith step
  - $prog$: the complete program

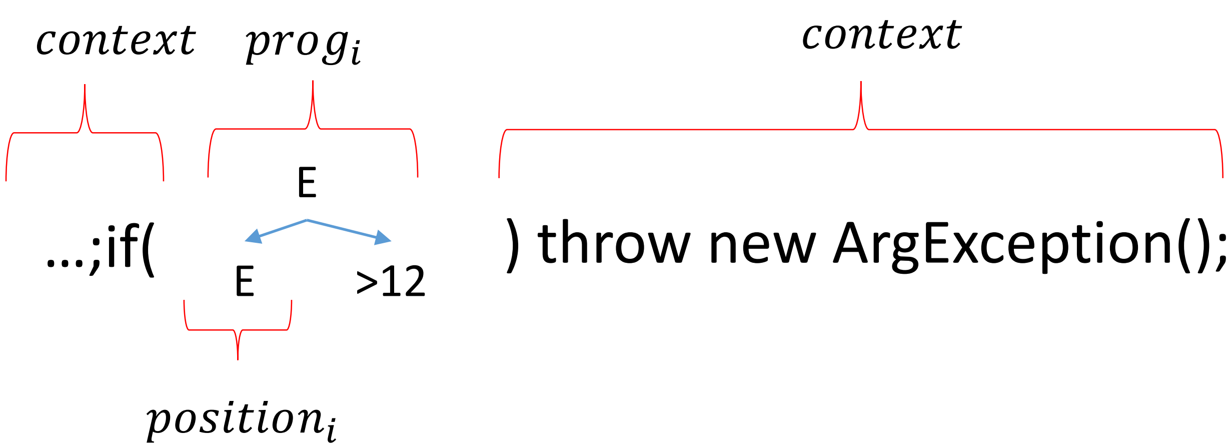


# Training models

- Train a model for each non-terminal
  - to classify rules expanding this non-terminal
- Training set preparation
  - The original training set:
    - A set of programs
    - Their contexts
  - Decomposing the training set:
    - Parse the programs
    - Extract the rules chosen for each non-terminal

# Feature Engineering

- Extract features from
  - $context$ : The context
  - $prog_i$ : The generated partial AST
  - $position_i$ : The position of the node to be expanded



# Can we choose non-leftmost nonterminal?

E

E > V

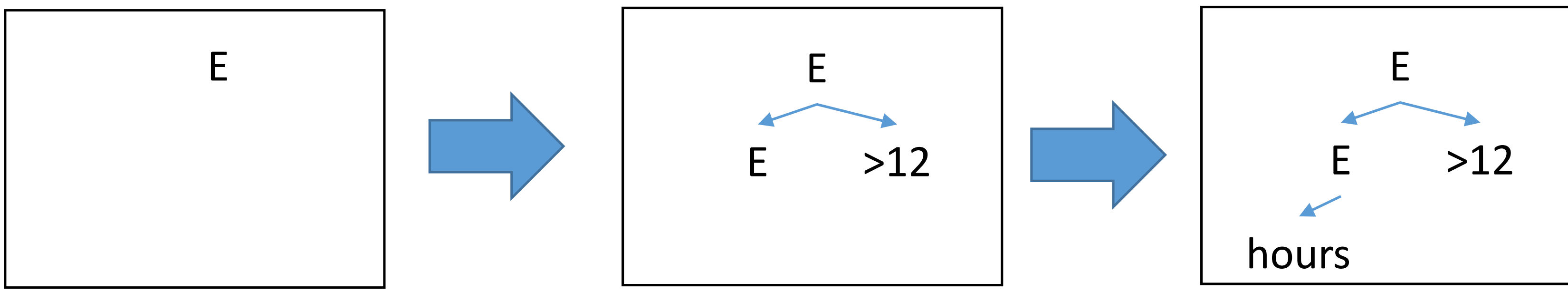
- If expanding V gives us more confidence, can we expand V first?

- Yes. We still have

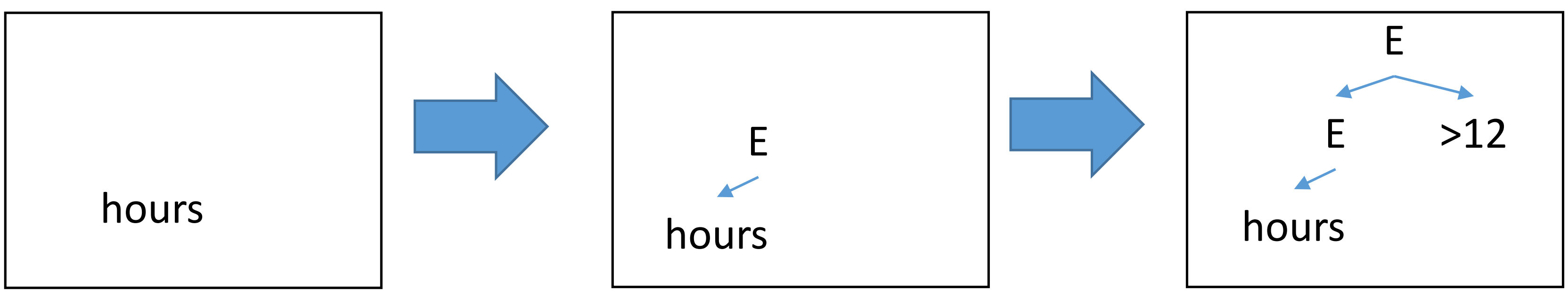
$$P(prog \mid context) = \prod_i P(\ rule_i \mid context, prog_i, position_i\ )$$

# Can we use a different expansion order?

- Top-down




- Bottom-up




The order may greatly affect the performance of L2S.

# Annotations

- Introduce annotations to symbols
  - $E^{D}$ indicates $E$ can be expanded downward
  - $E^{U}$ indicates $E$ can be expanded upward
  - $E^{UD}$ indicates $E$ can be expanded in both directions

# From Grammar to Rewriting Rules

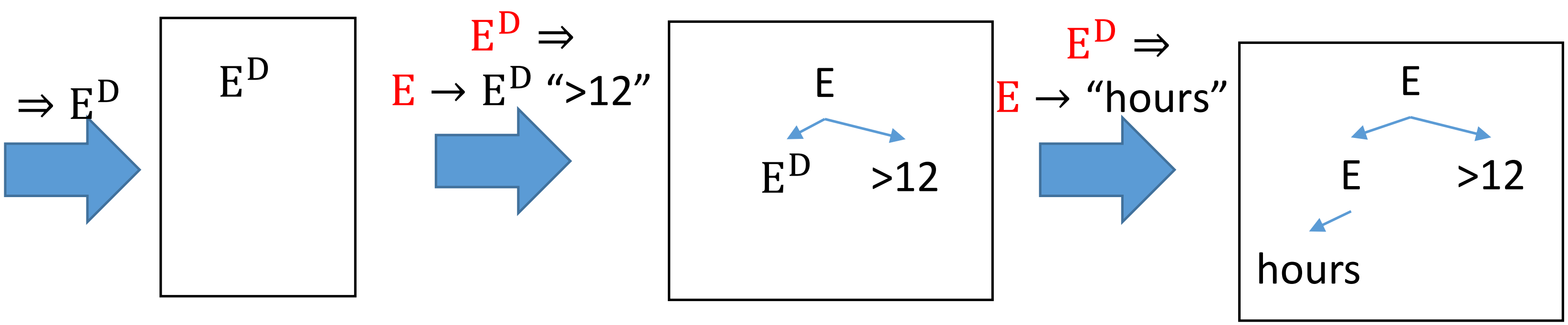
| Grammar | Top-down Rules | Bottom-up Rules |
|---|---|---|
| $E \rightarrow E$ "+" $E$ | $E^D \Rightarrow E \rightarrow E^D$ "+" $E^D$ | $E^U \Rightarrow E^U \rightarrow E$ "+" $E^D$<br>$E^U \Rightarrow E^U \rightarrow E^D$ "+" $E$ |
| $E \rightarrow E$ ">12" | $E^D \Rightarrow E \rightarrow E^D$ ">12" | $E^U \Rightarrow E^U \rightarrow E$ ">12" |
| $E \rightarrow$ "hours" | $E^D \Rightarrow E \rightarrow$ "hours" | "hours"$^U \Rightarrow E^U \rightarrow$ "hours" |

| Creation Rules | |
|---|---|
| $\Rightarrow E^D$ | // starting from the root |
| $\Rightarrow E^{DU}$ | // starting from a middle node |
| $\Rightarrow$ "hours"$^U$ | // starting from a leaf |

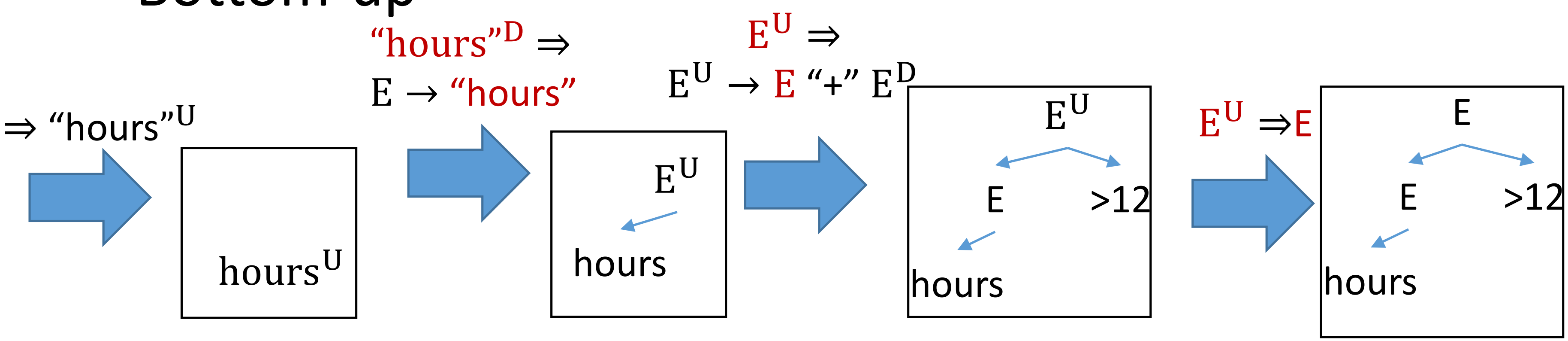
| Ending Rule | $E^U \Rightarrow E$ |
|---|---|

# Example

- Top-down




- Bottom-up



# Unambiguity

- A set of rewriting rules are unambiguous if
  - there is at most one unique set of rule applications to construct any program.

- When the rule set is unambiguous, we still have
  - $P(prog \mid context) = \prod_i P(\ rule_i \mid context, prog_i, position_i\ )$

# Challenge 2: How to find the most probable program?

- Local Optimal $\neq$ Global Optimal

$E_0$

| | |
|---|---|
| E → E " > 12" | 0.3 |
| E → E " > 0" | 0.6 |

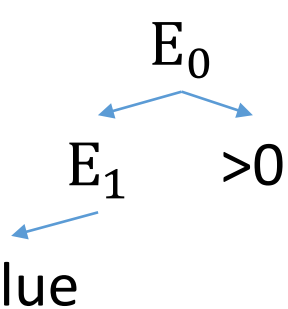
$E_1$

| | |
|---|---|
| E → "hours" | 0.1 |
| E → "value" | 0.2 |
| E → E " + " E | 0.05 |

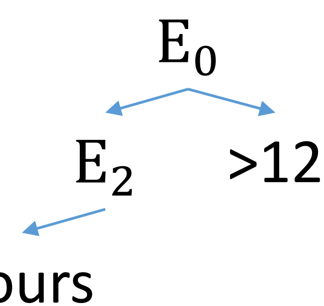
$E_2$

| | |
|---|---|
| E → "hours" | 0.8 |
| E → "value" | 0.1 |
| E → E " + " E | 0.05 |

0.6 * 0.2
= 0.12



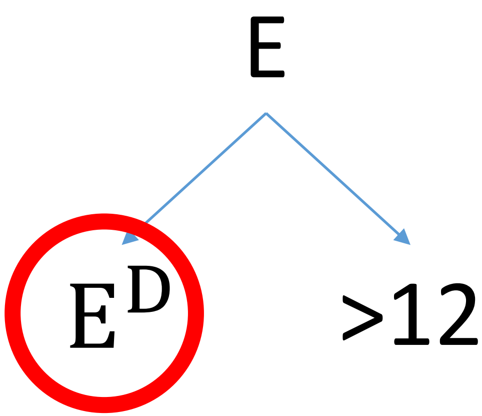

0.3 * 0.8
= 0.24



# Idea 1: Use Metaheuristic Search

- Beam Search:
  - Keep n most probable partial programs
  - Expand the programs to get new programs
- Genetic Search:
  - Keep n most probably complete programs
  - Mutate the programs to get new programs

# Idea 2: Pruning off Invalid Choices




$$
\begin{aligned}
E^D \Rightarrow\ & E \rightarrow E^D \text{ “+” } E^D \\
\mid\ & \cancel{E \rightarrow E^D \text{ “>12”}} \\
\mid\ & E \rightarrow \text{“hours”}
\end{aligned}
$$

- Generating constraints from the partial AST
  - Type constraints
  - Size constraints
  - Semantic constraints from E
- Use a solver to determine invalid choices

# Summary

- L2S Combines four tools
  - **Rewriting rules**: defining a search problem
  - **Constraint solving**: pruning off invalid choices in each step
  - **Machine-learned models**: estimating the probabilities of choices in each step
  - **Search algorithms**: solving the search problem

# Evaluation

- Evaluation 1:
  - Repairing Conditional Expressions
- Evaluation 2:
  - Generating Code from Natural Language Expression

# Repairing Conditional Expressions

- Condition bugs are common

```
  hours = convert(value);
+ if (hours > 12)
+   throw new ArithmeticException();
```

Missing boundary checks

```
-  if (hours >= 24)
+ if (hours > 24)
     withinOneDay=true;
```

Conditions too weak or too strong

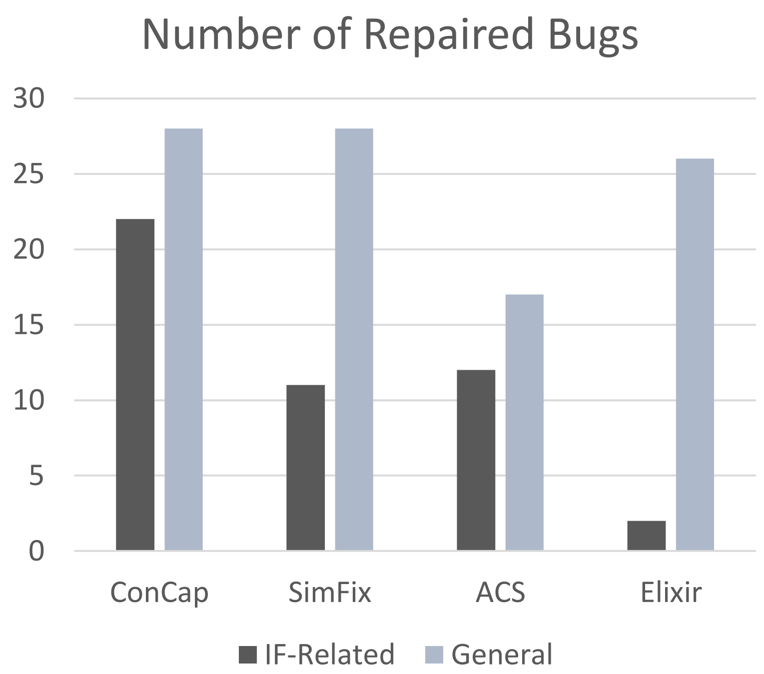
- Steps:
  1. Localize a buggy if condition with SBFL and predicate switching
  2. Synthesize an if condition to replace the buggy one
  3. Validate the new program with tests

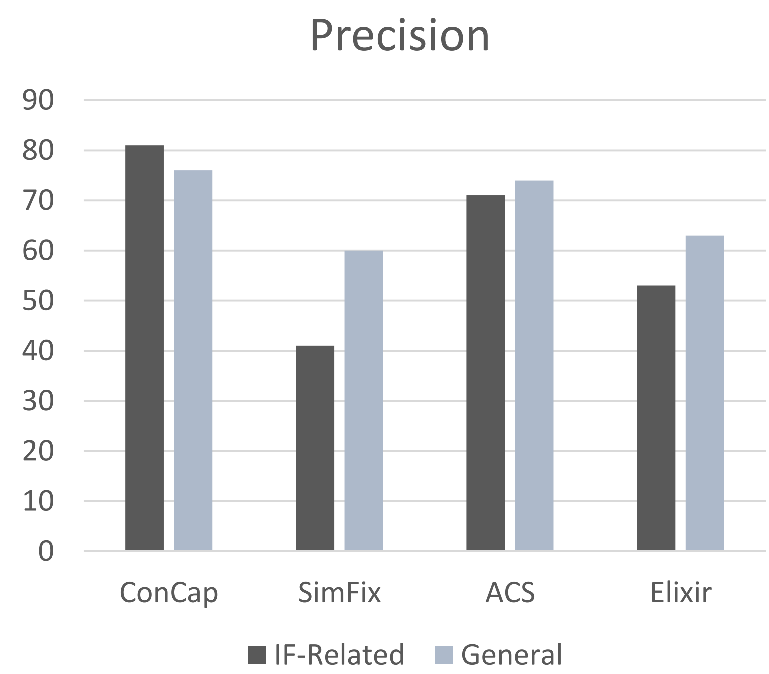
# L2S Configuration

- Rewriting rules
  - Bottom-up
  - Estimate the leftmost variable first
- Machine learning
  - Xgboost
  - Manually designed features
- Constraints
  - Type constraints & size constraints
- Search algorithm
  - Beam search

# Results

Benchmark: Defects4J




Also repaired 8 unique bugs that have never been repaired by any approach.

# Generating Code from Natural Language Expression

- Can we generate code automatically to avoid repetitive coding?
- Existing approaches use RNN to translate natural language descriptions to programs
  - **Long dependency problem**: work poorly on long programs


```
[NAME]
Acidic Swamp Ooze
[ATK] 3
[DEF] 2
[COST]  2
[DUR]  -1
[TYPE]  Minion
[CLASS] Neutral
[RACE] NIL
[RARITY] Common
[DESCRIPTION]
"Battlecry: Destroy Your Opponent's Weapon"
```

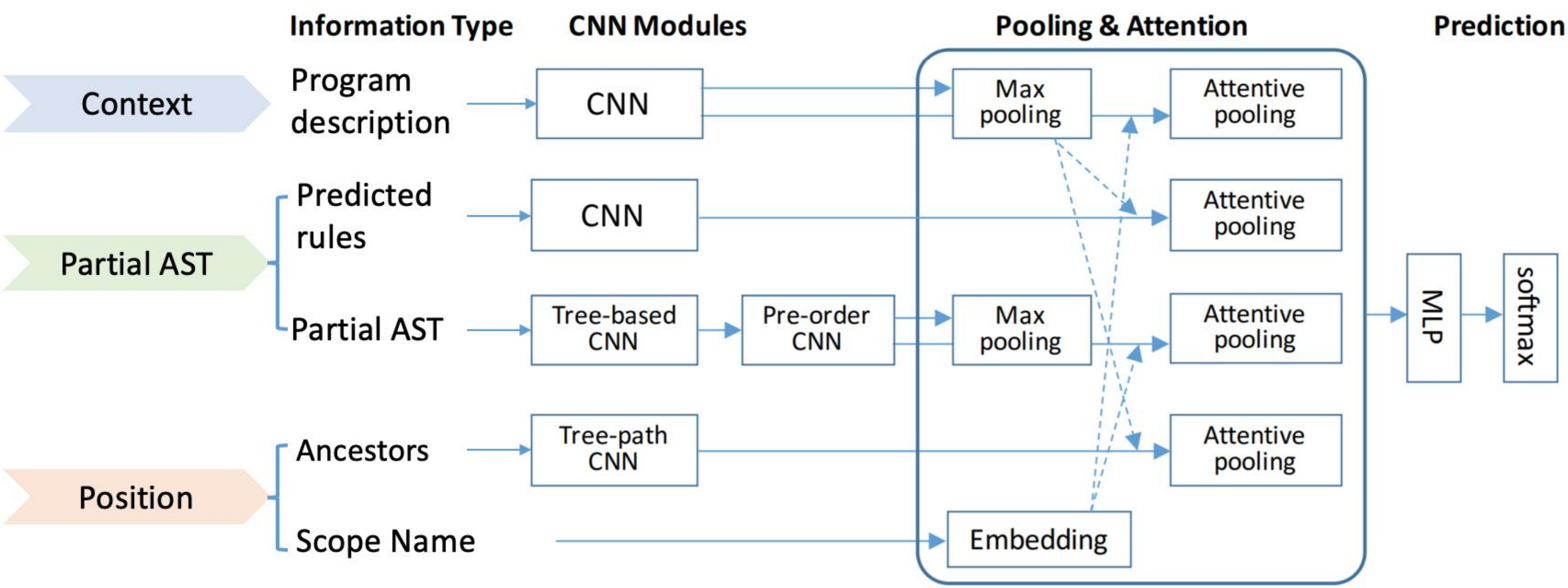
```python
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry=Battlecry(Destroy(), WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```

# L2S Configuration

- Rewriting rules
  - Top-down
- Machine learning
  - A CNN-based network
- Constraints
  - Size constraints
- Search algorithm
  - Beam search

# A CNN-based Network Architecture



# Results

Benchmark: HearthStone

| Model | StrAcc | Acc+ | BLEU |
|---|---|---|---|
| LPN (Ling et al. 2016) | 6.1 | – | 67.1 |
| SEQ2TREE (Dong and Lapata 2016) | 1.5 | – | 53.4 |
| SNM (Yin and Neubig 2017) | 16.2 | ∼18.2 | 75.8 |
| ASN (Rabinovich, Stern, and Klein 2017) | 18.2 | – | 77.6 |
| ASN+SUPATT (Rabinovich, Stern, and Klein 2017) | 22.7 | – | 79.2 |
| Our system | **27.3** | **30.3** | **79.6** |

# Conclusion

- Program Estimation: to find the most probable program under a context
- L2S: combining four tools to solve program estimation
- Why worked?
  - Machine learning to estimate probability
  - Rewriting rules and constraints to confine the space
  - Search algorithms to locate the best program
- Better to combine the tools we have

# Thank you for listening!

Main References:

[1] Yingfei Xiong, Bo Wang, Guirong Fu, Linfei Zang. Learning to Synthesize. GI'18: Genetic Improvment Workshop, May 2018

[2] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, Lu Zhang. A Grammar-Based Structural CNN Decoder for Code Generation. AAAI'19: Thirty-Third AAAI Conference on Artificial Intelligence, January 2019.