Our Goal

- We want to cause the program to fail
- We have seen
  - random (unstructured) input
  - structured (grammar-based) input
  - generation based on grammar coverage
A Challenge

class Roots {
    // Solve $ax^2 + bx + c = 0$
    public roots(double a, double b, double c) {
        ... }

    // Result: values for x
    double root_one, root_two;
}

• Which values for $a, b, c$ should we test?

    assuming $a, b, c$, were 32-bit integers, we’d have $(2^{32})^3 \approx 10^{28}$ legal inputs
    with $1.000.000.000.000$ tests/s, we would still require $2.5$ billion years
The Code

// Solve $ax^2 + bx + c = 0$
public roots(double a, double b, double c) {
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        // code for handling one root
    }
    else {
        // code for handling no roots
    }
}

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Test this case

and this

and this!
The Test Cases

// Solve \( ax^2 + bx + c = 0 \)
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        // code for handling one root
    }

    else {
        // code for handling no roots
    }
}
A Defect

// Solve $ax^2 + bx + c = 0$
public roots(double a, double b, double c) {
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {  // code for handling no roots
        // code for handling no roots
    }
}

$(a, b, c) = (0, 0, 1)$

(code must handle $a = 0$)
Use the program to guide test generation
The Ingredients

- Dynamic Coverage
- Static Structure
- Smart Algorithms
The Ingredients

- Dynamic Coverage
- Static Structure
- Smart Algorithms
Expressing Structure

// Solve $ax^2 + bx + c = 0$
public roots(double a, double b, double c) {
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }
}
Control Flow Graph

- A control flow graph expresses paths of program execution.
- Nodes are basic blocks – sequences of statements with one entry and one exit point.
- Edges represent control flow – the possibility that the program execution proceeds from the end of one basic block to the beginning of another.
public roots(double a, double b, double c)

double q = b * b - 4 * a * c;

q > 0 && a != 0

// code for two roots
q == 0

// code for one root
// code for no roots

return

- The CFG can serve as an *adequacy criterion* for test cases

- The more parts are covered (executed), the higher the chance of a test to uncover a defect

- “parts” can be: nodes, edges, paths, conditions…
Control Flow Patterns

while (COND)

 BODY

 do {
   BODY
 } while (COND);

for (INIT; COND; INCR)

 BODY;
/**
 * @title cgi_decode
 * @desc
 * Translate a string from the CGI encoding to plain ascii text
 * '+' becomes space, %%xx becomes byte with hex value xx,
 * other alphanumeric characters map to themselves
 * returns 0 for success, positive for erroneous input
 * 1 = bad hexadecimal digit
 */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
while (*eptr) /* loop to end of string (‘\0’ character) */
{
    char c;
    c = *eptr;
    if (c == '+') /* ‘+’ maps to blank */
        *dptr = ' ';
    } else if (c == '%') /* ‘%xx’ is hex for char xx */
    {
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
        else
            *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}

*dptr = ‘\0’; /* Null terminator for string */
return ok;
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    while (*eptr) {
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
            ++dptr;
            ++eptr;
            break;
        } else if (c == '%') {
            ++eptr;
            ++dptr;
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        } else {
            *dptr = *eptr;
            ++dptr;
        }
    }
    *dptr = '0';
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    while (*eptr)
    {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        }
        else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
        }
        ++dptr;
        ++eptr;
    }
    *dptr = '\0';
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';  // G
            ++dptr;
            ++eptr;
            continue;
        }
        else if (c == '%') {
            *dptr = *eptr;  // F
            ++dptr;
            *dptr = *eptr;  // F
            ++dptr;

            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;  // I
                ++dptr;
                ++eptr;
                continue;
            }
            *dptr = 16 * digit_high + digit_low;  // H
        }
        else {
            *dptr = *eptr;  // E
            ++dptr;
            ++eptr;
        }
    }
    *dptr = '\0';
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded) {
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
        }
        ++dptr;
        ++eptr;
    }
    *dptr = "0";
    return ok;
}
```
def cgi_decode(encoded, decoded):
    eptr = encoded
    dptr = decoded
    ok = 0
    c = eptr
    while c:
        if c == '+':
            dptr = ' ';
        else:
            if c == '%':
                dptr = *eptr;
            else:
                digit_high = Hex_Values[(++eptr)];
                digit_low = Hex_Values[(++eptr)];
                if (digit_high == -1 || digit_low == -1):
                    ok = 1;
                else:
                    *dptr = 16 * digit_high + digit_low;
        ++dptr;
        ++eptr;
    *dptr = "0";
    return ok;
Test Adequacy Criteria

- How do we know a test suite is "good enough"?
- A test adequacy criterion is a predicate that is true or false for a pair \( \langle \text{program, test suite} \rangle \)
- Usually expressed in form of a rule – e.g., "all statements must be covered"
Statement Testing

- Adequacy criterion: each statement (or node in the CFG) must be *executed at least once*

- Rationale: a defect in a statement can only be revealed by *executing* the defect

- Coverage: $\# \text{executed statements} \over \# \text{statements}$
```c
int cgi_decode(char *encoded, char *decoded) {
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    int digit_high = Hex_Values[*(++eptr)];
    int digit_low = Hex_Values[*(++eptr)];
    if (digit_high == -1 || digit_low == -1) {
        ok = 1;
    } else {
        *dptr = 16 * digit_high + digit_low;
    }
    ++dptr;
    ++eptr;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';  // F
    } else if (c == '%') {
        *dptr = *eptr;  // C
    } else {
        *dptr = *eptr;
    }
    while (*eptr) {
        ++dptr;
        ++eptr;
        int digit_high = Hex_Values[*++eptr];
        int digit_low = Hex_Values[*++eptr];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
        }
        if (c == '+') {
            *dptr = ' ';  // F
        } else if (c == '%') {
            *dptr = *eptr;  // C
        } else {
            *dptr = *eptr;
        }
    }
    *dptr = '0';
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';  
    }
    while (*eptr) {
        True
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';  
        }
        else if (c == '%') {
            *dptr = *eptr;
        }
        else {
            *dptr = '0';
            return ok;
        }
        ++dptr;
        ++eptr;
        }else {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        }
    } else {
            *dptr = '0';
            return ok;
        }
}  
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr)
    {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
            break;
        } else if (c == '%') {
            *dptr = *eptr;
            ++dptr;
            ++eptr;
            continue;
        } else {
            *dptr = *eptr;
            ++dptr;
            ++eptr;
        }
    }

    int digit_high = Hex_Values[*(++eptr)];
    int digit_low = Hex_Values[*(++eptr)];
    if (digit_high == -1 || digit_low == -1) {
        ok = 1;
    } else {
        ok = 1;
        *dptr = 16 * digit_high + digit_low;
    }

    *dptr = "0";
    return ok;
}
```
Computing Coverage

• Coverage is computed automatically while the program executes

• Requires *instrumentation* at compile time
  With GCC, for instance, use options `-ftest-coverage -fprofile-arcs`

• After execution, *coverage tool* assesses and summarizes results
  With GCC, use “gcov source-file” to obtain readable .gcov file
int ok = 0;

while (*eptr) /* loop to end of string ('\0' character) */
{
    char c;
    c = *eptr;
    if (c == '+') { /* '+' maps to blank */
        *dptr = ' ';
    } else if (c == '%') { /* '%xx' is hex for char xx */
        int digit_high = Hex_Values[*++eptr];
        int digit_low = Hex_Values[*++eptr];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
        else
            *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}

*dptr = '\0'; /* Null terminator for string */
return ok;
And now…

Let’s build our own coverage tools!
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' '
        elif c == '%':
            digit_high = s[i + 1]
            digit_low = s[i + 2]
            i = i + 2
            if (digit_high in hex_values and
                digit_low in hex_values):
                v = (hex_values[digit_high] * 16 +
                     hex_values[digit_low])
                t = t + chr(v)
        else:
            raise Exception
    else:
        raise Exception
    return t
Python Tracing

• In Python, tracing executions is much simpler than in compiled languages.
• The function `sys.settrace(f)` defines `f()` as a *tracing function* that is invoked for every line executed.
• `f()` has access to the *entire interpreter state*.
```python
import sys

def traceit(frame, event, arg):
    if event == "line":
        lineno = frame.f_lineno
        print("Line", lineno, frame.f_locals)
    return traceit

sys.settrace(traceit)
```

**Python Tracing**

current frame (PC + variables)

import sys

def traceit(frame, event, arg):
    if event == "line":
        lineno = frame.f_lineno
        print("Line", lineno, frame.f_locals)
    return traceit

sys.settrace(traceit)

“line”, “call”, “return”, …

tracer to be used in this scope (this one)

https://docs.python.org/2/library/sys.html?highlight=settrace#sys.settrace
Demo
The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms
The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms
Coverage Goals

• With dynamic coverage, we can find out all statements executed (also branches and paths, if we track pairs or lists of lines)
• But how do we know the set of possible statements?
• Need to analyze program statically
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' ' 
        elif c == '%':
            ...
        else:
            t = t + c
        i = i + 1
    return t
Python AST

- The Python AST module converts a Python source file into an abstract syntax tree
- The tree can be traversed using a visitor pattern
Python AST

import ast

root = ast.parse('x = 1
print(ast.dump(root))

AST root

Python input

AST as string (for debugging)

https://docs.python.org/2/library/ast.html#ast.AST
```python
import ast

root = ast.parse('x = 1')
print ast.dump(root)

→
Module(
    body = [
        Assign(
            targets = [
                Name(id = 'x', ctx = Store())
            ],
            value = Num(n=1)
        )
    ]
)  
```
Demo
AST Visitor

- The `ast.NodeVisitor` class provides a `visit(n)` method which traverses all subnodes of `n`.
- Should be subclassed to be extended.
- On each node `n` of type `TYPE`, the method `visit_TYPE(n)` is called if it exists.
- If there is no `visit_TYPE(n)`, the method `generic_visit()` traverses all children.
class IfVisitor(ast.NodeVisitor):
    def visit_If(self, node):
        print("if", node.lineno, ":")
        for n in node.body:
            print("    ", n.lineno)
        print "else:"
        for n in node.orelse:
            print("    ", n.lineno)
        self.generic_visit(node)
root = ast.parse(open('cgi_decode.py').read())

v = IfVisitor()
v.visit(root)
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' ' 
        elif c == '%':
            digit_high = s[i + 1]
            digit_low = s[i + 2]
            i = i + 2
            if (digit_high in hex_values and
                digit_low in hex_values):
                v = (hex_values[digit_high] * 16 +
                     hex_values[digit_low])
                t = t + chr(v)
            else:
                raise Exception 
        else:
            t = t + c
            i = i + 1
    return t
Demo
The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms
The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms
Approaches

- Random Testing: *ignore* program structure
- Symbolic Testing: *solve* path conditions leading to uncovered statements
- Search-Based Testing: still random, but have structure *guide* test generation
Evolutionary Algorithms

Create population

Create mutations

Rank

Select

Recombine (optional)
Evolutionary Algorithms

Create population

“fdsakfh+ew%3gfhd4f”
“fwe8^ru786234jä”

Mutate

“fdsakfh+br%3gfhd4f”
“fdsakfh+ew%4gfhd4f”
“fwe8^ru&26234jä”
“xb3#ru786234jä”
Evolutionary Algorithms

Mutate

“fdsakfh+br%3gfhd%4f”
“fdsakfh+ew%4gfhd%4f”

“fwe8^ru&26234jä”
“xb3#ru786234jä”

Recombine

“fdsakfh+ew%4gfhd%4f”

“xb3#ru786234jä”
Evolutionary Algorithms

Mutate

“fdsakfh+br%3gfhd%4f”
“fdsakfh+ew%4gfhd%4f”
“fwe8^ru&26234jà”
“xb3#ru786234jà”

Recombine

“fdsakfh+ew%4gfhd%4f”
“xb3#ru786234jà”
“xb3#akfh+ew%4gfhd%4f”
Selection and Ranking

if (angle = 47 ∧ force = 532) { ... }

angle = 51

“xb3#ru786234jä”

angle = 48

angle = 47
Evolutionary Algorithms

- Create population
- Create mutations
- Rank
- Select
- Recombine (optional)
And now...

Let’s implement this in Python!
The General Plan

- Create a population of random inputs
- Obtain their coverage
- Higher coverage = higher fitness
  (a bit simplistic, but will do the job)
- Select individuals with high fitness
  (say, the 25% fittest individuals)
- Mutate them to obtain offspring
The Mutation Plan

• For each input, keep a history of the grammar productions that lead to it

$$START \rightarrow \$EXPR \rightarrow \$TERM \rightarrow \$FACTOR \rightarrow \$INTEGER \rightarrow \$DIGIT \rightarrow 2$$

• To mutate, truncate that history and apply different productions from there on

$$START \rightarrow \$EXPR \rightarrow \$TERM \rightarrow \$FACTOR \rightarrow \$INTEGER \rightarrow \$DIGIT \rightarrow 2$$
$$FACTOR \rightarrow \$INTEGER \rightarrow \$DIGIT \rightarrow 4$$
CGI Grammar

cgi_grammar = {
    "$START": ["$STRING"],
    "$STRING": ["$CHARACTER", "$STRING$CHARACTER"],
    "$CHARACTER": ["$REGULAR_CHARACTER", "$PLUS", "$PERCENT"],
    "$REGULAR_CHARACTER": ["a", "b", "c", ".", ":", ":!"],
    # actually more
    "$PLUS": ["+"]
    "$PERCENT": ["%$HEX_DIGIT$HEX_DIGIT"],
    "$HEX_DIGIT": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", 
                    "a", "b", "c", "d", "e", "f"]
}
Demo
Evolution Cycle

pop = population(grammar)

for i in range(EVOLUTION_CYCLES):
    # Evolve the population
    print("Evolved:")
    next_pop = evolve(pop, grammar)
    print_population(next_pop)
    pop = next_pop
# Create a random population

def population(grammar):
    pop = []
    while len(pop) < POPULATION_SIZE:
        try:
            # Create a random individual
            term, productions = produce(cgi_grammar)
        except AssertionError:
            # Try again
            continue

        # Determine its fitness (by running the test, actually)
        fitness = coverage_fitness(term)

        # Add it to the population
        pop.append((term, productions, fitness))

    return pop
# Where we store the coverage
coverage = {}

# Now, some dynamic analysis
def traceit(frame, event, arg):
    global coverage
    if event == "line":
        lineno = frame.f_lineno
        # print("Line", lineno, frame.f_locals)
        coverage[lineno] = True
    return traceit

# Define the fitness of an individual term – by actually testing it
def coverage_fitness(term):
    # Set up the tracer
    global coverage
    coverage = {}
    sys.settrace(traceit)

    # Run the function under test
    result = cgi_decode(term)

    # Turn off the tracer
    sys.settrace(None)

    # Simple approach:
    # The term with the highest coverage gets the highest fitness
    return len(coverage.keys())
def by_fitness(individual):
    (term, production, fitness) = individual
    return fitness

# Evolve the set
def evolve(pop, grammar):
    # Sort population by fitness (highest first)
    best_pop = sorted(pop, key=by_fitness, reverse=True)

    # Select the fittest individuals
    best_pop = best_pop[:SELECTION_SIZE]

    # Breed
    offspring = []
    while len(offspring) + len(best_pop) < POPULATION_SIZE:
        parent = random.choice(best_pop)
        child = mutate(parent, grammar)

        (parent_term, parent_productions, parent_fitness) = parent
        (child_term, child_productions, child_fitness) = child

        if child_fitness >= parent_fitness:
            offspring.append(child)

    return offspring
# Evolve the set
def evolve(pop, grammar):
    # Sort population by fitness (highest first)
    best_pop = sorted(pop, key=by_fitness, reverse=True)

    # Select the fittest individuals
    best_pop = best_pop[:SELECTION_SIZE]

    # Breed
    offspring = []
    while len(offspring) + len(best_pop) < POPULATION_SIZE:
        parent = random.choice(best_pop)
        child = mutate(parent, grammar)

        (parent_term, parent_productions, parent_fitness) = parent
        (child_term, child_productions, child_fitness) = child

        if child_fitness >= parent_fitness:
            offspring.append(child)

    next_pop = best_pop + offspring

    # Keep it sorted
    next_pop = sorted(next_pop, key=by_fitness, reverse=True)

    return next_pop
# Create a mutation from PARENT, generating one offspring

def mutate(parent, grammar):
    (parent_term, parent_productions, parent_fitness) = parent

    # Truncation cutoff: only keep CUTOFF productions
    cutoff = random.randint(0, len(parent_productions) - 1)

    # Repeat the first CUTOFF production steps of parent
    child_term = "$START"
    child_productions = []
    for i in range(cutoff):
        rule = parent_productions[i]
        child_term = apply_rule(child_term, rule)
        child_productions.append(rule)

    # From here on, proceed in random direction
    extra_productions = None
    while extra_productions is None:
        try:
            child_term, extra_productions = produce(grammar, child_term)
        except AssertionError:
            pass  # Just try again
# Repeat the first CUTOFF production steps of parent
child_term = "$START"
child_productions = []
for i in range(cutoff):
    rule = parent_productions[i]
    child_term = apply_rule(child_term, rule)
    child_productions.append(rule)

# From here on, proceed in random direction
extra_productions = None
while extra_productions is None:
    try:
        child_term, extra_productions = produce(grammar, child_term)
    except AssertionError:
        pass  # Just try again

child_productions += extra_productions

# Compute its fitness
child_fitness = coverage_fitness(child_term)

print("Mutated " + repr(parent_term) + " to " + repr(child_term))

# And we're done
return child_term, child_productions, child_fitness
## Populations

(with fitness)

<table>
<thead>
<tr>
<th>Initial</th>
<th>After 20 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'+%60c!a%08' 16</code></td>
<td><code>'+%60c!a%08' 16</code></td>
</tr>
<tr>
<td><code>'%8fc+%8da.+16</code></td>
<td><code>'%8fc+%8da.+16</code></td>
</tr>
<tr>
<td><code>'+%f2!' 16</code></td>
<td><code>'+%f2!' 16</code></td>
</tr>
<tr>
<td><code>'b%26%d2%60' 15</code></td>
<td><code>'+%80a.+16</code></td>
</tr>
<tr>
<td><code>'%f6b' 15</code></td>
<td><code>'%9fc+%8da.+16</code></td>
</tr>
<tr>
<td><code>'b%f2' 15</code></td>
<td><code>'%61+%f7b' 16</code></td>
</tr>
<tr>
<td><code>'.%c5' 15</code></td>
<td><code>'+%21b.+16</code></td>
</tr>
<tr>
<td><code>'%60+' 15</code></td>
<td><code>'%1c%04+%a3.+16</code></td>
</tr>
<tr>
<td><code>'%87' 14</code></td>
<td><code>'%7b++c.+16</code></td>
</tr>
<tr>
<td><code>'%1a' 14</code></td>
<td><code>'+l%fa+%21a.+16</code></td>
</tr>
<tr>
<td><code>'%53' 14</code></td>
<td><code>'+%ca+%71.+16</code></td>
</tr>
<tr>
<td><code>'%77' 14</code></td>
<td><code>'+%60c!a%08' 16</code></td>
</tr>
<tr>
<td><code>'+.' 10</code></td>
<td><code>'%e0b+a' 16</code></td>
</tr>
<tr>
<td><code>'+!+.' 10</code></td>
<td><code>'%99c+%8da.+16</code></td>
</tr>
<tr>
<td>`!' 9</td>
<td><code>'%d4++%8ca.+16</code></td>
</tr>
<tr>
<td>`':.' 9</td>
<td><code>'%20a+%f7b' 16</code></td>
</tr>
<tr>
<td>`'+.' 8</td>
<td><code>'++%f2a' 16</code></td>
</tr>
<tr>
<td>`'+.' 8</td>
<td><code>'%95c+' 16</code></td>
</tr>
<tr>
<td>`'++.' 8</td>
<td><code>'%7fc+%8da.+16</code></td>
</tr>
<tr>
<td>`'++.' 8</td>
<td><code>'++%f2!' 16</code></td>
</tr>
</tbody>
</table>
Things to do

• Use a *derivation tree* to represent both inputs and histories (much more efficient)

• Use a *genetic algorithm* with *recombination* rather than only mutation

• Base fitness function on *approach level* – how close are we to a yet uncovered line?

• Integrate code and grammar coverage…