Fuzzing with Mutants

Security Testing

Andreas Zeller
Rahul Gopinath
Saarland University
Fuzzing with Mutants and Grammars

Security Testing

Andreas Zeller
Rahul Gopinath
Saarland University
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>

Start fuzzing from what we know as legal inputs
Christian Holler
The Road to Success
The Road to Success

1. Use a grammar to create inputs
The Road to Success

1. Use a grammar to *create* inputs
2. Use a grammar to *parse* inputs
The Road to Success

1. Use a grammar to create inputs
2. Use a grammar to parse inputs
3. Mutate and recombine inputs
Parsing
n = mult(x, 2);

Parsing
n = mult(x, 2);

Parsing
Mutating

expression
=

variable
= 
n

mult
x 
2

call
Mutating
n = div(x, 2);

Mutating
Recombining
z = y << 3;
z = y << 3;

n = mult(x, 2);

Recombining
Recombining
Recombining

z = 2;

expression

= expression

variable

2

z

n = mult(x, y << 3);

expression

= expression

variable

mult

<<

<<

x

y

3

<<

n

mult

x

<<

y

3

<<
$z = 2;$

$\text{expression} = \text{variable} \ 2$

$n = \text{mult}(x, x \ll 3);$  

$\text{expression} = \text{variable} \ = \ \text{call} \ \text{mult} \ \text{expression}$

Recombining + Renaming
Recombining + Renaming

\[ z = 2; \]

\[ n = \text{mult}(x, x \ll 3); \]
The Road to Success

1. Use a grammar to create inputs
2. Use a grammar to parse inputs
3. Mutate and recombine inputs
The Road to Success

1. Use a grammar to *create* inputs
2. Use a grammar to *parse* inputs
3. *Mutate* and *recombine* inputs
4. Use a *bug repository* as source for samples
<table>
<thead>
<tr>
<th>ID</th>
<th>Product</th>
<th>Component</th>
<th>Assignee</th>
<th>Status</th>
<th>Resolution</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1260288</td>
<td>Thunderbird</td>
<td>Security</td>
<td>aclists</td>
<td>UNCO</td>
<td>---</td>
<td>Changed server not reflected in Saved password list</td>
</tr>
<tr>
<td>2396690</td>
<td>Thunderbird</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>Allow for advanced challenge/response authentication (optical flickering)</td>
</tr>
<tr>
<td>309886</td>
<td>Core</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>after changing the character-encoding, access to JavaScript functions is blocked</td>
</tr>
<tr>
<td>394041</td>
<td>Core</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>purge of unsolicited restricted site cookies on Mozilla startup (RealPlayer 10 sets cookies in Mozilla when browser is not executing)</td>
</tr>
<tr>
<td>327493</td>
<td>Thunderbird</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>Support timestamp protocol (TSP) as per RFC3161</td>
</tr>
<tr>
<td>358620</td>
<td>Core</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>Installing a signed XPI with OCSP enabled causes &quot;internal failure&quot; error</td>
</tr>
<tr>
<td>383703</td>
<td>Thunderbird</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>pop mail server authentication issue for users with two mail accounts from different service providers</td>
</tr>
<tr>
<td>413650</td>
<td>Thunderbird</td>
<td>Security</td>
<td>dvdeltz</td>
<td>UNCO</td>
<td>---</td>
<td>offer to import encryption certificates and not yet trusted root certificates (e.g. if in the encryptionKeyPrefSet is set)</td>
</tr>
<tr>
<td>1064208</td>
<td>Developer Documentation</td>
<td>API Miscellaneous</td>
<td>eshepherd</td>
<td>UNCO</td>
<td>---</td>
<td>Remove netscape security PrivilegeManager from Windowsstatusbar</td>
</tr>
<tr>
<td>487209</td>
<td>Testopia</td>
<td>Security</td>
<td>greganh</td>
<td>UNCO</td>
<td>---</td>
<td>Allow selection of a Bzbugzilla Group with Testopia's Access Control List</td>
</tr>
<tr>
<td>530749</td>
<td>Testopia</td>
<td>Security</td>
<td>greganh</td>
<td>UNCO</td>
<td>---</td>
<td>When selecting Product Dashboard in Testopia - I get Not authorised error - same in FF35.5 and IE9</td>
</tr>
<tr>
<td>559976</td>
<td>Testopia</td>
<td>Security</td>
<td>greganh</td>
<td>UNCO</td>
<td>---</td>
<td>The test run assignee should be able to update test runs w/o requiring write perm to the plan</td>
</tr>
<tr>
<td>583011</td>
<td>Testopia</td>
<td>Security</td>
<td>greganh</td>
<td>UNCO</td>
<td>---</td>
<td>Test plan &quot;User Regular Expression&quot; definitions apply only retroactively, and not proactively</td>
</tr>
<tr>
<td>598217</td>
<td>Testopia</td>
<td>Security</td>
<td>greganh</td>
<td>UNCO</td>
<td>---</td>
<td>User can see all Test Runs and Cases not only those for which products, he has access to</td>
</tr>
<tr>
<td>576332</td>
<td>Penelope</td>
<td>General</td>
<td>mduziak</td>
<td>UNCO</td>
<td>---</td>
<td>in Eudora cannot cancel message window about invalid security certificate</td>
</tr>
<tr>
<td>769352</td>
<td>mozilla.org</td>
<td>Security Assurance</td>
<td>mgoodin</td>
<td>UNCO</td>
<td>---</td>
<td>SeeReview: make tab strip async</td>
</tr>
<tr>
<td>224765</td>
<td>MailNews Core</td>
<td>Security</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>Feature Request: A list of mail addresses to which SMIME signatures should not be sent</td>
</tr>
<tr>
<td>294249</td>
<td>Firefox</td>
<td>Security</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>Request for a secure site tray/tabtasker type Notice/Advisory screen and notification icon.</td>
</tr>
<tr>
<td>318594</td>
<td>Toolkit</td>
<td>XUL Widgets</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>NS_ERROR_DOM_SECURITY_ERR for message.xml</td>
</tr>
<tr>
<td>3323550</td>
<td>Thunderbird</td>
<td>Security</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>emails flagged as possible Email Scam should be rendered as simple html (similar to mail:spam.display.size)</td>
</tr>
<tr>
<td>344904</td>
<td>Core</td>
<td>Security</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>Restrict ajax/javascript scope to DOM element.</td>
</tr>
<tr>
<td>351919</td>
<td>Toolkit</td>
<td>Startup and Profile</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>startup reports 'Could not initialize the browser's security component'</td>
</tr>
<tr>
<td>360752</td>
<td>NSS</td>
<td>Libraries</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>netsign S5320 Security Device causes crashes</td>
</tr>
<tr>
<td>356049</td>
<td>NSS</td>
<td>Libraries</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>Device Manager fails to add a security module after it has been deleted</td>
</tr>
<tr>
<td>377657</td>
<td>Firefox</td>
<td>Security</td>
<td>nobody</td>
<td>UNCO</td>
<td>---</td>
<td>HTTP authentication warning box shows up with href attribute but not with src attribute</td>
</tr>
</tbody>
</table>
Fuzzing with Code Fragments

Authors:
Christian Höller, Mozilla Corporation; Kim Hartig and Andreas Zeller, Saarland University

Abstract:
Fuzz testing is an automated technique providing random data as input to a software system in the hope to expose a vulnerability. In order to be effective, the fuzzed input must be common enough to pass elementary consistency checks; a JavaScript interpreter, for instance, would only accept a semantically valid program. On the other hand, the fuzzed input must be uncommon enough to trigger exceptional behavior, such as a crash of the interpreter. The LangFuzz approach resolves this conflict by using a grammar to randomly generate valid programs, the code fragments, however, partially stem from programs known to have caused invalid behavior before. LangFuzz is an effective tool for security testing. Applied on the Mozilla JavaScript interpreter, it discovered a total of 165 new severe vulnerabilities within three months of operation (and thus became one of the top security bug bounty collectors within this period); applied on the PHP interpreter, it discovered 18 new defects causing crashes.

Open Access Media
USENIX is committed to Open Access to the research presented at our events. Papers and proceedings are freely available to everyone once the event begins. Any video, audio, and/or slides that are posted after the event are also free and open to everyone. Support USENIX and our commitment to Open Access.
The Road to Success

1. Use a grammar to create inputs
2. Use a grammar to parse inputs
3. Mutate and recombine inputs
The Road to Success

1. Use a grammar to \textit{create} inputs \textcolor{red}{- done}
2. Use a grammar to \textit{parse} inputs
3. \textit{Mutate} and \textit{recombine} inputs
The Road to Success

1. Use a grammar to create inputs – done
2. Use a grammar to parse inputs
3. Mutate and recombine inputs – exercise
The Road to Success

1. Use a grammar to *create* inputs— done
2. Use a grammar to *parse* inputs— ???
3. *Mutate* and *recombine* inputs— exercise
Parsing
n = mult(x, 2);

Parsing
\[ n = \text{mult}(x, 2); \]

**Parsing**
Parsing

• Classical problem of computer science (in particular compiler construction)

• Writing good parsers is difficult:
  • Generality – work on wide set of languages
  • Efficiency – parse in linear time
  • These goals may be mutually exclusive
Two Parsers
Two Parsers

- We discuss two parsers:
Two Parsers

- We discuss two parsers:
  1. A *general one* (but rather inefficient)
Two Parsers

• We discuss two parsers:
  1. A *general one* (but rather inefficient)
  2. An *efficient one* (but not as general)
Two Parsers

• We discuss two parsers:
  1. A general one (but rather inefficient)
  2. An efficient one (but not as general)
• Both turn a string into a derivation tree
Two Parsers

• We discuss two parsers:
  1. A *general one* (but rather inefficient)
  2. An *efficient one* (but not as general)

• Both turn a string into a *derivation tree*

• The derivation tree can then be used for mutation, recombination, etc.
A General Parser

1. Have a queue of derivation trees sorted by number of nonterminals (initially $START$)

2. Take tree $t$ from queue

3. Enqueue all expansions of $t$ that fit the input (e.g. prefix of terminals)

4. Continue at 2. until full match is found
Demo
# Parse input INP into a derivation tree

def parse(inp, grammar, start_symbol = "$START"):

    # The queue of symbols to try out
    queue = [init_tree(grammar, start_symbol)]

    while len(queue) > 0:
        tree = queue[0]  # Retrieve first element from queue
        queue = queue[1:]

        # Check whether the expansion still matches
        prefix = all_terminals_prefix(tree)

        if prefix is not None:
            if len(prefix) > 0:
                if not inp.startswith(prefix):
                    # No match: Discard and try next alternative
                    continue
                if inp == prefix:
                    # Derivation tree matches – we’re done
                    return tree
# Check whether the expansion still matches
prefix = all_terminals_prefix(tree)

if prefix is not None:
    if len(prefix) > 0:
        if not inp.startswith(prefix):
            # No match: Discard and try next alternative
            continue
        if inp == prefix:
            # Derivation tree matches – we're done
            return tree

# Find expansions for the leftmost nonterminal
expansions = leftmost_nonterminal_expansions(tree, grammar)

# Try them next
queue += expansions

# Keep queue sorted by length of expressions, # thus always trying shorter rules first
queue.sort(key = lambda tree: possible_expansions(tree))

# Tried out all alternatives
return None
An Efficient Parser

Sacrifice *generality* for *efficiency*!
An Efficient Parser

1. Match(Token)
   1. For any terminal, match by equality with the input.
   2. For any non-terminal, fetch the rules to match, and return the first match.

2. Match(Rule)
   1. A rule matches if all tokens in the rule can be matched with the input tokens in sequence.

Match(input, $START)
Context Free Grammar

Characteristics:

• Unambiguous
• Non-deterministic

PALINDROM := 0 S 0
  | 1 S 1
  | ε

e.g: 011101101110

Best possible parsing:

O(N^3)  Generalized LR Parser
Deterministic CFG

INT := INT DIGIT
   | DIGIT

DIGIT := 1 | 2 | 3 | .... 0

Characteristics:

• Unambiguous
• Deterministic
• Left Recursive
Ambiguous CFG

EXPR := EXPR + EXPR
   | EXPR - EXPR
   | INT

INT := INT DIGIT
   | DIGIT

DIGIT := 1 | 2 | 3 | .... 0

1 - 2 + 3 => 1 - (2 + 3)
   (1 - 2) + 3

Characteristics:
- Ambiguous
- Deterministic
- Left Recursive

Parsers
- Top Down
- Bottom Up
Deterministic CFG

EXPR     := EXPR + FACTOR
    | FACTOR
FACTOR:= FACTOR * INT
    | INT
INT     := INT DIGIT
    | DIGIT
DIGIT := 1 | 2 | 3 | .... 0

1 * 2 + 3 => (1 * 2) + 3
1 + 2 * 3 => 1 + (2 * 3)

Characteristics:

• Unambiguous
• Deterministic
• Left Recursive

Parsers

• Top Down
• Bottom Up
Deterministic CFG

A simple algorithm

```python
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res:
            return (res, l)
    return (False, 0)
```

EXPR  :=  EXPR  +  FACTOR  
       |  FACTOR

FACTOR := FACTOR * INT  
         |  INT

INT    := INT DIGIT  
         |  DIGIT

DIGIT  := 1 | 2 | 3 | .... 0

1 * 2 + 3 => (1 * 2) + 3
1 + 2 * 3 => 1 + (2 * 3)
Deterministic CFG

EXPR : = FACTOR + EXPR
      | FACTOR

FACTOR : = INT * FACTOR
        | INT

INT : = DIGIT INT
     | DIGIT

DIGIT : = 1 | 2 | 3 | .... 0

1 * 2 + 3 => (1 * 2) + 3
1 + 2 * 3 => 1 + (2 * 3)

A simple algorithm

def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res:
            return (res, l)
    return (False, 0)
Parsing Expression Grammar

expr  ::= factor + expr | factor

factor ::= int * factor | int

int    ::= digit int | digit

digit ::= 1 | 2 | 3 | .... 0

1 * 2 + 3 => (1 * 2) + 3
1 + 2 * 3 => 1 + (2 * 3)

Packrat algorithm

def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res:
            return (res, l)
    return (False, 0)
Parsing Expression Grammar

EXPR     := FACTOR + EXPR  
          |  FACTOR

FACTOR:= INT * FACTOR  
          |  INT

INT      := DIGIT INT  
          |  DIGIT

DIGIT := 1 | 2 | 3 | .... 0

1 * 2 + 3 => (1 * 2) + 3
1 + 2 * 3 => 1 + (2 * 3)

Packrat algorithm

def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res:
            return (res, l)
    return (False, 0)

First available
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].startswith(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res:
            return (res, l)
    return (False, 0)
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

IF (a = b) THEN return 1 ELSE return 0
IF (a = b) THEN return 1
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

IFS := IF $C$ THEN $S$ ELSE $S$
/$\quad$ IF $C$ THEN $S$

Tradeoff:
• Ordered choice

IF (a = b) THEN return 1 ELSE return 0
IF (a = b) THEN return 1
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

IF (a = b) THEN return 1 ELSE return 0
IF (a = b) THEN return 1

Tradeoff:
• Ordered choice

$S := \text{IF} \ C \ \text{THEN} \ \$S \ \text{ELSE} \ \$S$
/ \ \text{IF} \ C \ \text{THEN} \ \$S$
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

Tradeoff:
- Ordered choice

\[ S := \text{IF} \ C \text{THEN} \ S \]
\[ / \text{IF} \ C \text{THEN} \ S \text{ELSE} \ S \]
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

IF (a = b) THEN return 1
IF (a = b) THEN return 1 ELSE return 0
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

IF (a = b) THEN return 1
IF (a = b) THEN return 1 ELSE return 0

Trading Expression Grammar

Tradeoff:
- Ordered choice

$S := IF $C THEN $S$
/ IF $C$ THEN $S$ ELSE $S$
def match_rule(rule, text, at):
    if literal(rule):
        return (text[at:].starts_with(rule), len(rule))
    tokens = split(rule)
    for token in tokens:
        result, l = match_token(token, text, at)
        if not result:
            return False
        at += l
    return (True, len)

def match_token(token, text, at):
    if seen((token, text, at)):
        return old_result
    rules = grammar[token]
    for rule in rules:
        res, l = match_rule(rule, text, at)
        if res: return (res, l)
    return (False, 0)

$S := IF $C THEN $S$
/ IF $C THEN $S ELSE $S$

Tradeoff:
• Ordered choice

IF (a = b) THEN return 1
IF (a = b) THEN return 1 ELSE return 0

✓
✗
Parsing Expression Grammar

Characteristics:

- Ordered choice
- Arrange the grammar such that
  - Longest rules are first

```latex
$S := IF \; C \; \text{THEN} \; S \; \text{ELSE} \; S$
\begin{align*}
&\text{/ IF } \; C \; \text{THEN} \; S
\end{align*}
```

- Any Non-Left Recursive grammar can be automatically converted to PEG.
Demo
Your Exercise

1. Use and extend our parsers to implement grammar-based mutation
Handouts
# Parse input INP into a derivation tree

```python
def parse(inp, grammar, start_symbol = "$START"):
    # The queue of symbols to try out
    queue = [init_tree(grammar, start_symbol)]

    while len(queue) > 0:
        tree = queue[0]  # Retrieve first element from queue
        queue = queue[1:]

        # Check whether the expansion still matches
        prefix = all_terminals_prefix(tree)

        if prefix is not None:
            if len(prefix) > 0:
                if not inp.startswith(prefix):
                    # No match: Discard and try next alternative
                    continue
                if inp == prefix:
                    # Derivation tree matches – we're done
                    return tree
```
# Find expansions for the leftmost nonterminal
expansions = leftmost_nonterminal_expansions(tree, grammar)

# Try them next
queue += expansions

# Keep queue sorted by length of expressions,
# thus always trying shorter rules first
queue.sort(key = lambda tree: possible_expansions(tree))

# Tried out all alternatives
return None