Automated Testing, and Generating Complex Input
(and other applications)

Course Software Testing & Verification

2022/23

Wishnu Prasetya & Gabriele Keller

Content

- What is “complex input”
- Regular expression and CFG/BNF to describe complex inputs.
  - Generating complex inputs
  - Coverage over the structure of complex inputs

Note: this lecture is about describing, generating, and covering complex inputs. We will focus on complex inputs that are described by means of grammars (regular or context free). This subject is only partially addressed by A&O. E.g. chapter 5 (2ed. Ch 9), set up the right background, but then they went on to focus more on mutation.

Complex Inputs

- Complex inputs are inputs (of a program) that are required to satisfy a non-trivial constraint.
- Examples:
  1. Prog(List s) where s has to be a sorted list.
  2. Prog(String s) where s has to be a string containing a valid email address.
- In this lecture we will focus on complex inputs whose constraints can be expressed by a ”grammar”.

Complex Inputs

- We discussed automated testing: consider a program P(x) with an in-code specification Q. Having an in-code spec. allows you to automatically test P(x) by generating x → good!
- Now, suppose x has to satisfy a non-trivial constraint C (e.g. it has to be a string representing an email address).
- Challange-1: simply randomly generating x may not be an effective way to produce x's.
- Challenge-2: it may also be necessary to make sure that we cover various corner cases of C. How to define a suitable concept of coverage over this C? (note that this is may be orthogonal to coverage over P's code)

Regular expressions as input constraints

- For example P(String s) where s is a phone number, e.g. 030 2530011
- We can use a regular expression to express this constraint:

PhoneNumber = 0*PD Space PDDDDDD*
D = 0|*P*
P = 1|…|9
Space = ␣ *

- (red symbols are terminals)

Another example: NL post codes

NLpostcode = Area Space Street
Area = PDDD
Street = Letter Letter
P = 1 | 2 ...
D = 0 | *P*
Letter = a | b | c ... | A | B | C ...
Space = ␣ *

***Note**: in practice there may be additional constraints, e.g. codes above 9999 XL do not actually exists (do not map to an existing address). This may require additional filtering which we will not go into in the lecture.*

Regular expression

Syntax:

rexp ::= *terminal*
or *rexp* | *rexp*
or *rexp* *rexp*
or *rexp**
or *rexp*+
or (*rexp*)

L(*e*) = the set of strings/sentences described by the rexp *e*, defined as follows:

L(*terminal*) = { *terminal* }
L(*e* | *f*) = **L**(*e*) ∪ **L**(*f*)
L(*ef*) = { *s*++*t* | *s*∈**L**(*e*), *t*∈**L**(*f*) }
L(*e**) = { ε } ∪ **L**(*e*+)
L(*e*+) = **L**(*ee**)

Using FSM to describe a language

Let M = $(s_0, f, V, E, \sum \cup \{\tau\})$ be an FSM.

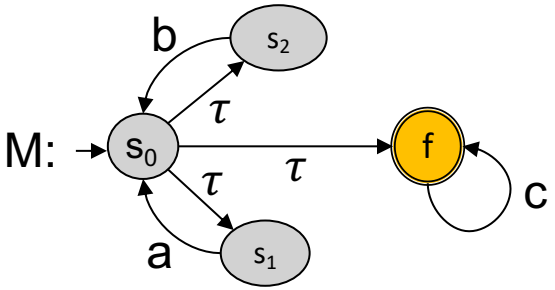
- τ is an "empty" label ("").
- $s_0 \in V$ is the initial state, $f \in V$ is the final state.

- An **execution** of M is a path σ in M, starting in s_0 and ends the final state f.
- The sequence of labels over $\sum$ (so, excluding τ) along an execution σ is called the **sentence** of σ.
- The **language described by M**, denoted by **L**(M), is the set of the sentences of all executions of M.

Equivalence between Rexp and FSM

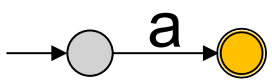
Let R be a regular expression with ∑ as its alphabet (its set of terminal symbols). There exists an FSM $M = (s_0, f, V, E, \sum \cup \{\tau\})$, such that **L**(R) = **L**(M).

Example: **(a|b)*c*** can be equivalently described by:

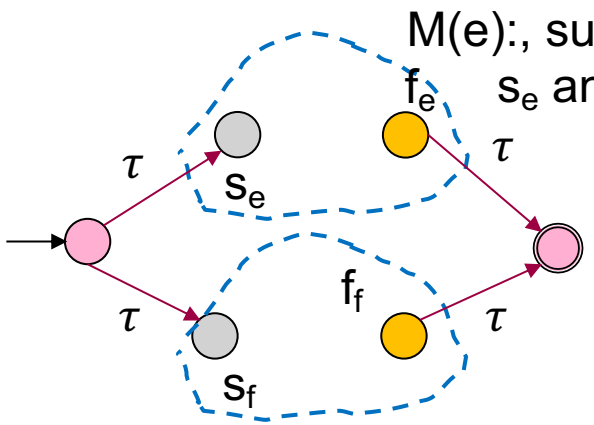


Converting Rexpr to FSM

An FSM describing the same language as a given regular expression can be recursively constructed as follows:

case-1 M(a): 

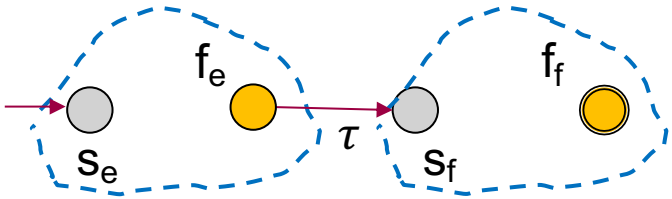
case-2 M(*e* | *f*):



Converting Rexpr to FSM

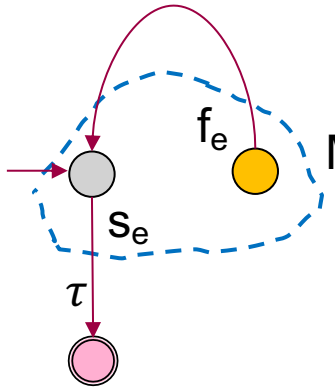
M(e):, suppose starting at s_e and ends at f_e.

case-3 M(*ef*):



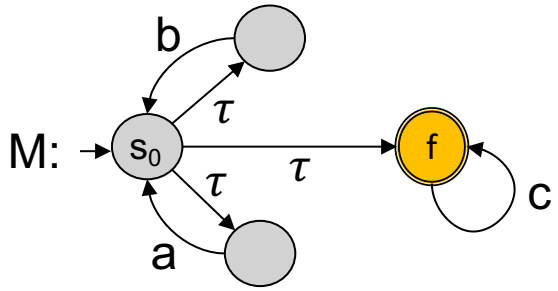
M(f):, suppose starting at s_f and ends at f_f.

case-4 M(*e**):



M(e):, suppose starting at s_e and ends at f_e.

Using FSM to generate tests

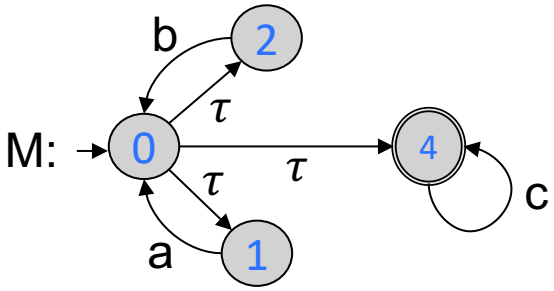


- It becomes easy to generate valid inputs, simply by following the FSM towards its end state. In other words, the FSM is essentially an input-generator.
- It also provides a (structural) concept of test coverage, namely the graph-based coverage that we already discussed (e.g. edge coverage, edge-pair coverage, or prime path coverage).

  Note: node/state coverage is typically too weak for FSM.

Example

Consider P(String s) where s is a string satisfying the regular expression: (a|b)*c*.

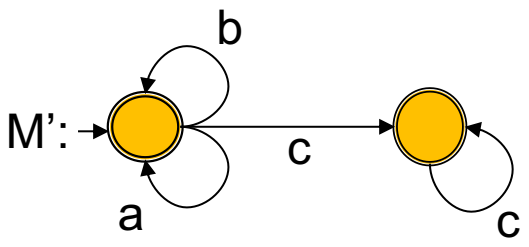


- Prime paths: 010, 101, 104, 102, 020, 202, 204, 201, 44
- A test set for P(s) that would give full PPC over the regular expression constraint of s:

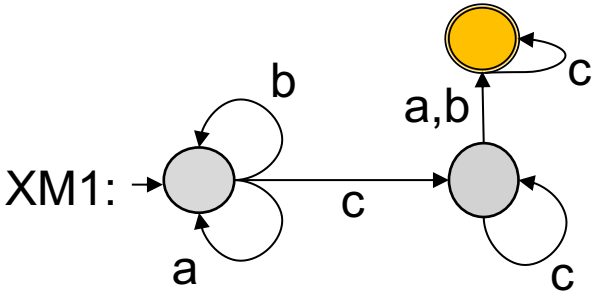
s	Prime path
aa	010, 101, 104
bb	020, 202, 204
abc	102, 204, 44
bac	201, 104, 44

Note that here we talk about coverage over P's input constraint, which may be an orthogonal concern with respect to e.g. the coverage over P's internal code.

Using FSM to generate negative tests



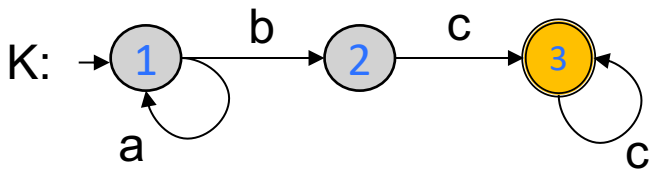
(deterministic variant of previous M)



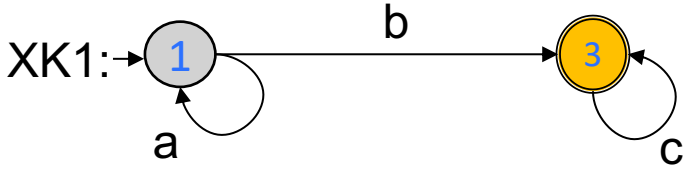
(a mutated M for generating invalid inputs)

- Negative test: test a program using an invalid input. Useful to check if the program handles such an input properly (e.g. should not crash, or should throw the right exception)
- We can mutate M to produce mutated versions (mutants) to systematically generate invalid inputs (note: we first reduce the FSM to make it deterministic) for performing negative testing.
- You may want mutants that produce inputs that are "slightly" wrong, rather than inputs that are just blatantly wrong, e.g. because this is where the program is more likely to go wrong.

Using FSM to generate negative tests



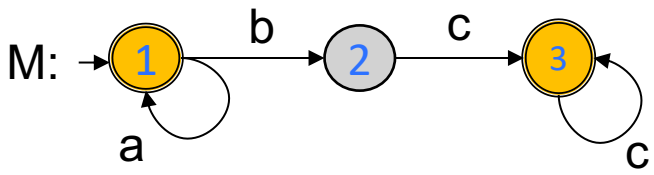
(original FSM)



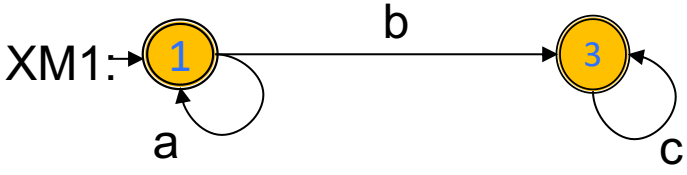
(a mutated K, bypassing state 2)

- As mutation operators you can think of:
  - changing the terminal state
  - adding transitions that were not possible in the original M
  - by-passing a state (see example above)
- Note: some care must be taken with mutants that can produce a valid sentence (of the original M). That is, a mutant XM where **L**(XM) ∩ **L**(M) is not empty.

Example of a mutant that is not disjoint



(original FSM)



(a mutated M obtained by bypassing state 2)

- XM1 is obtained by bypassing state-2 in M. But notice that L(XM1) ∩ L(M) is not empty. So, XM1 is not guaranteed to generate an invalid input for the original program.

  Not useless as long as L(XM1)/L(M) is not empty, though now it takes more effort to get an instance of an invalid input.

Limitation of regular expressions

- Consider P(String s) where s must be an HTML document.
- An HTML document is a sequence of elements, where each element E starts with <*E*> and ends with </*E*>.
- Unfortunately, such a pattern cannot be expressed by a regular expression, essentially because regular expressions cannot count.
- We will look at a more expressive way to express string constraint, namely *Context Free Grammar* (CFG).

Context Free Grammar

A context free grammar (CFG) consists of:

- a set of symbols called **terminals**.
- a set of symbols called **non-terminals**; one of it is special, called the "start symbol".
- a set of **production rules**. Each has the form N $\rightarrow$ Z, where N is a non-terminal and Z is a *sequence of symbols*.

It describes a **language** whose sentences are built from the CFG's terminals.
In A&O, CFG is also called Backus-Naur Form (BNF).

Example

$$
\begin{aligned}
\textbf{\textit{S}} &\rightarrow \textit{Brace} \mid \textit{Curly} \mid \varepsilon \\
\textit{Brace} &\rightarrow \text{“(“ } S \text{ “)” } S \\
\textit{Curly} &\rightarrow \text{“\{“ } S \text{ “\}” } S
\end{aligned}
$$

With **S** as the start symbol

- Examples of sentences allowed by this grammar are:

  ”()” ”{()} {}”

- Mismatched braces/curlies are rejected. E.g. “(})” is not a sentence accepted by this grammar.

Extra notation for production rule

- A rule like $A \rightarrow a(B|C)d$ is seen as a *short hand* for a set of production rules:

  $A \rightarrow aBd$
  $A \rightarrow aCd$

- People often use *extended BNF* e.g. :

  Brace $\rightarrow$ (“(“ *S* “)”)*

Deriving valid strings

$$
\begin{aligned}
S &\rightarrow \text{Brace} \mid \text{Curly} \mid \varepsilon \\
\text{Brace} &\rightarrow \text{“(“ } S \text{ “)” } S \\
\text{Curly} &\rightarrow \text{“\{“ } S \text{ “\}” } S
\end{aligned}
$$

A ***derivation*** *is a series of expansion of the grammar that result in a sequence of terminal symbols. It follows that the sequence is a valid sentence of the grammar. We can use this to generate valid sentences.* Example :

$$
\begin{aligned}
S &\rightarrow \text{Brace} \\
&\rightarrow (\ S\ )\ S \\
&\rightarrow (\ \varepsilon\ )\ S \\
&\rightarrow (\ \varepsilon\ )\ \varepsilon
\end{aligned}
$$

Let's first name the rules

Name	Prod. rule
RSB	$S \rightarrow Brace$
RSC	$S \rightarrow Curly$
RSE	$S \rightarrow \varepsilon$
RB	$Brace \rightarrow$ "(" S ")" S
RC	$Curly \rightarrow$ "{" S "}" S

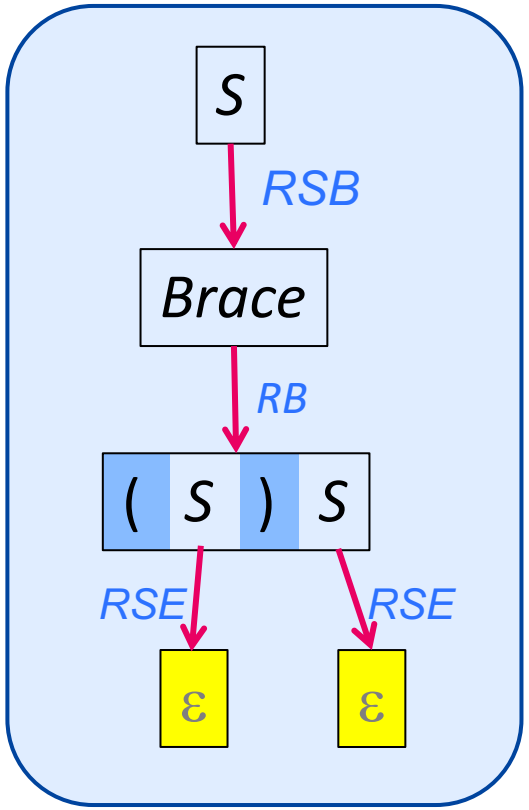
Derivation tree (instead of sequence)

Name	Prod. rule
RSB	$S \rightarrow$ *Brace*
RSC	$S \rightarrow$ *Curly*
RSE	$S \rightarrow \varepsilon$
RB	*Brace* $\rightarrow$ "(" S ")" S
RC	*Curly* $\rightarrow$ "{" S "}" S

A *derivation* ***sequence*** *of "( )"*:

$S \rightarrow$ *Brace*
$\rightarrow$ (S) S
$\rightarrow$ (ε) S
$\rightarrow$ (ε) ε

The *derivation* ***tree*** *of "( )"*:

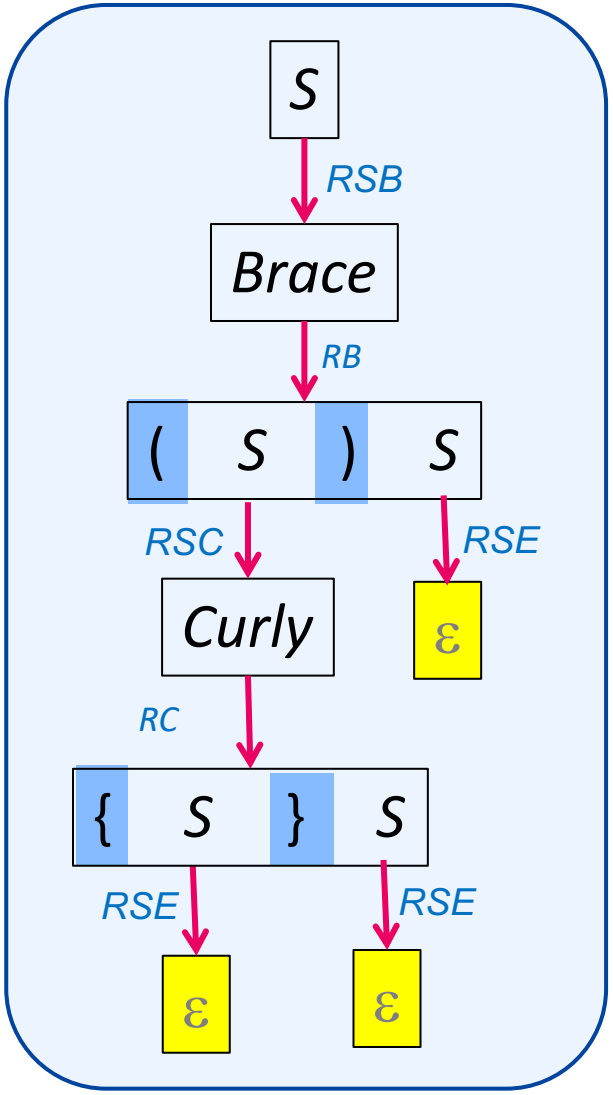


A derivation can also be described by a derivation tree such as above. Given such a tree, you can reconstruct what the derived sentence is.

One more example

The *derivation* ***tree*** *of* “({ })”:

Name	Prod. rule
RSB	*S* → *Brace*
RSC	*S* → *Curly*
RSE	*S* → ε
RB	*Brace* → “(“ *S* “)” *S*
RC	*Curly* → “{“ *S* “}” *S*

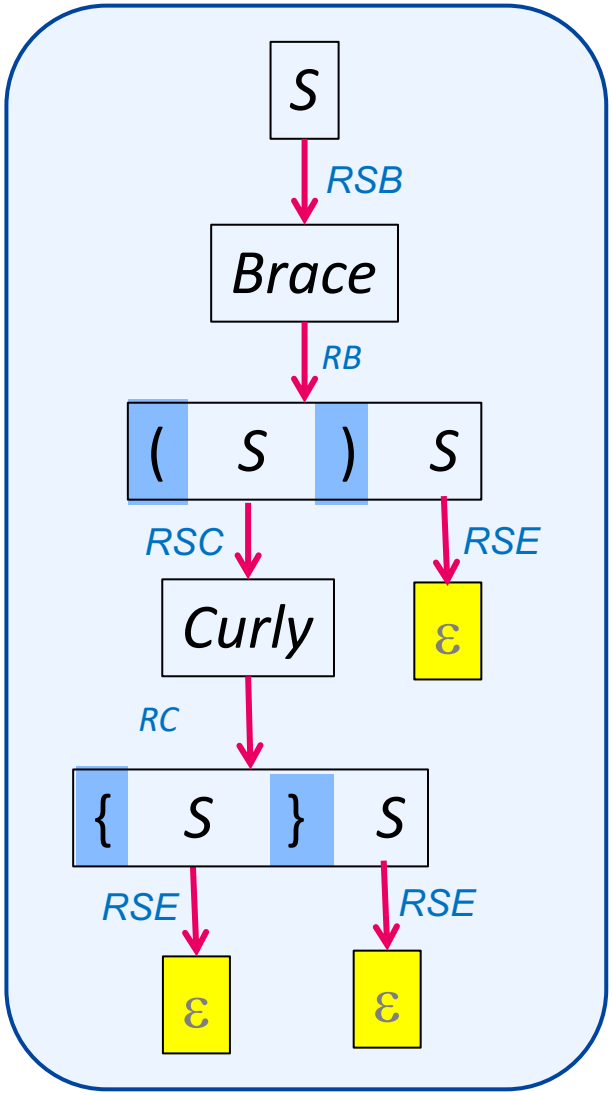


producing “({ })”

Generating valid/invalid strings through derivation

- Imagine a program P(s) where s is a string whose format has to satisfy a context free grammar G.
- Valid inputs can be generated by first generating derivation trees for G, e.g. randomly or exhaustively up to a certain depth.
- A negative/invalid input can be generated e.g. by generating a derivation tree from a mutated G', where we deliberately change one of G's production rule. Note however, that this may produce a sentence t that turns out to be in **L**(G). It is hard to know this upfront.
- Still to answer: a concept of coverage over G.

We can see...

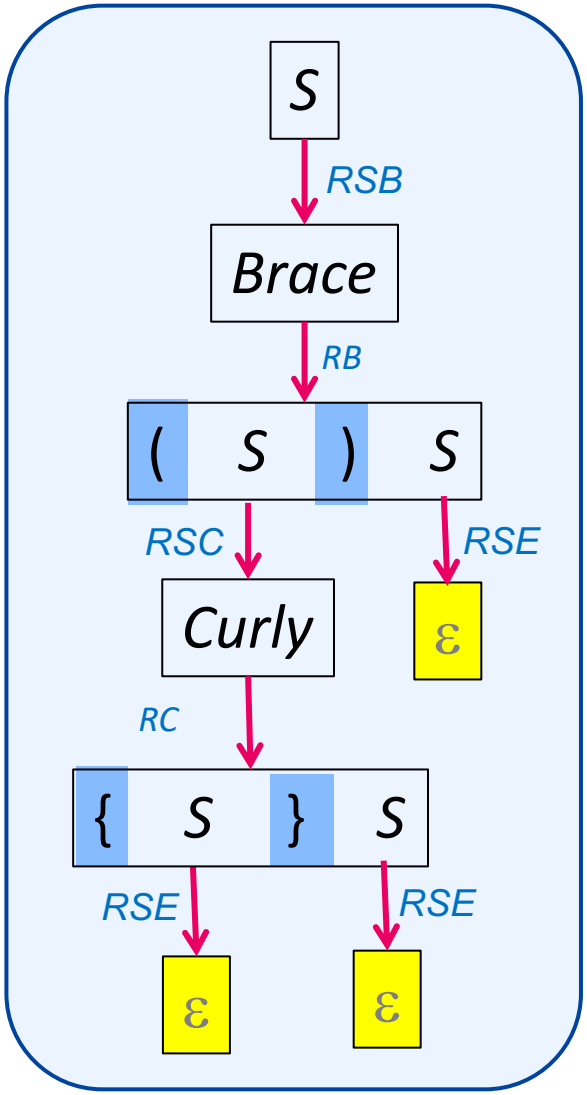


- We can see which non-terminals and terminals are produced by the derivation.
- We can see which rules were used.

CFG/BNF coverage

- (C5.29/2nd Ed. C9.31) TR contains each terminal symbol from the given grammar *G.*
- (C5.30/2nd Ed. C9.32) TR contains each production rule in G.
- Production rule coverage subsumes terminal coverage; but both are usually too weak. For example, the single test case from the previous slide covers all production rules of its grammar.
- *Pair-wise and k-wise rule coverage*
- *Rule-rule coverage*

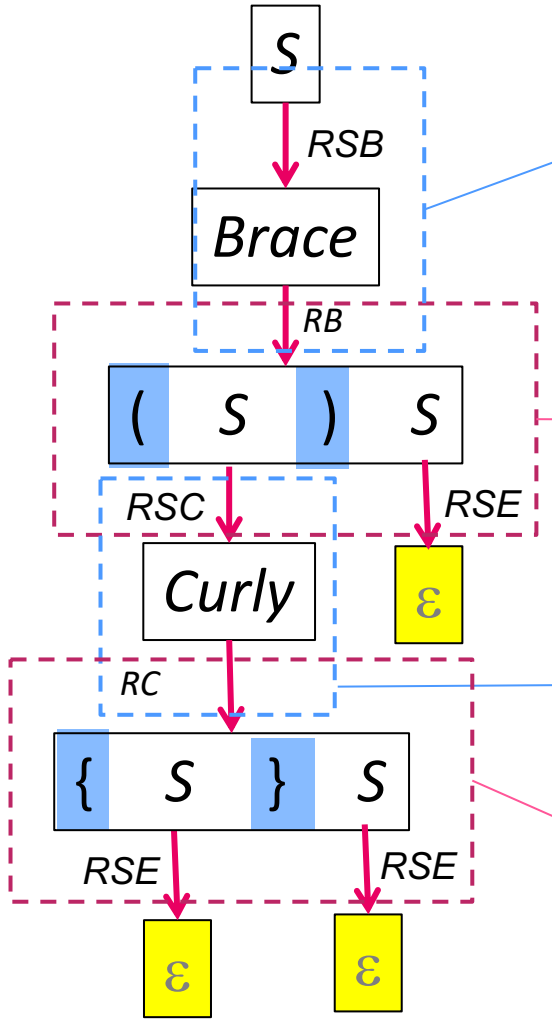
Pair-wise rule coverage



- Consider a CFG G.
- A derivation tree *t* of G covers covers a **pair** of production rules **<R_1;R_2>** if the pair appears as two *consecutive* arrows in in *t.* (note the order).
- A set *T* of derivation trees gives full pair-wise rule coverage if every *feasible* pair of rules <R_1;R_2> is covered by some *t* in *T*.
- Analogously we can define k-wise rule coverage.

Example of covering rule-pairs

The rule pairs covered by this single derivation:



covers the 1xpair **<RSB;RB>**

covers 2x pairs: **<RB;RSC>** and **<RB;RSE>**

covers 1x pair: **<RSC;RC>**

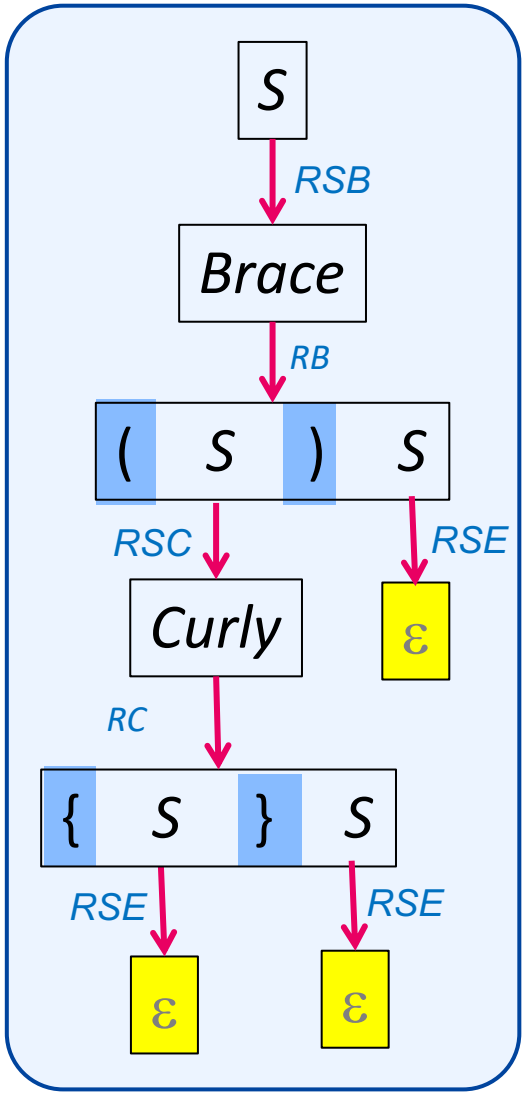
covers 1xpair: **<RC;RSE>**

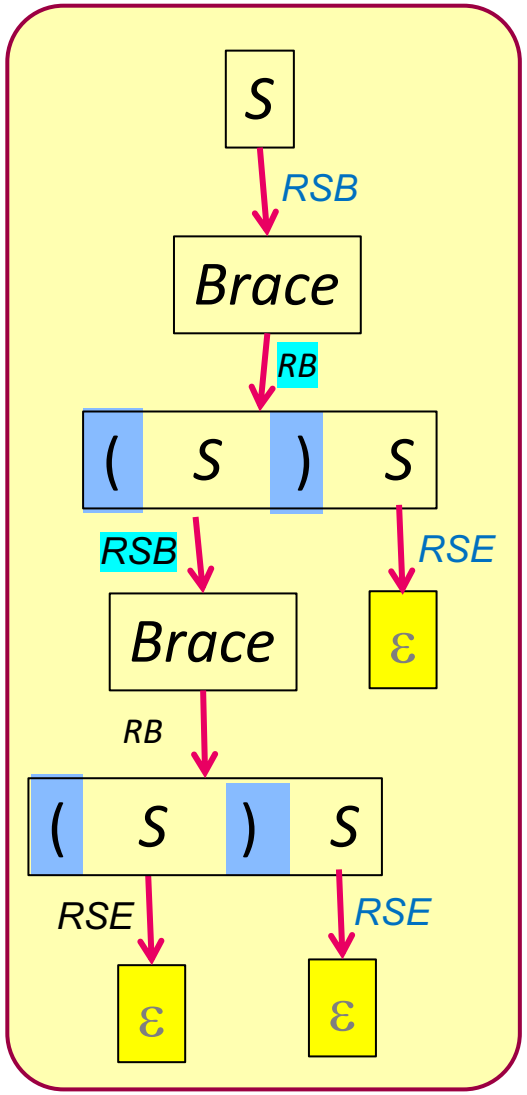
Feasible pairs

Name	Prod. rule	Feasible rule-pairs
RSB	*S* → *Brace*	<RSB;RB>
RSC	*S* → *Curly*	<RSC;RC>
RSE	*S* → ε	
RB	*Brace* → "(" *S* ")" *S*	<RB;RSB>, <RB;RSC>, <RB;RSE>
RC	*Curly* → "{" *S* "}" *S*	<RC;RSB>, <RC;RSC>, <RC,RSE>

- Only feasible pairs count.
- Let $R_1 = U \rightarrow Z_1$ and $R_2 = V \rightarrow Z_2$, where Z_1 and Z_2 are sequences of symbols that may contain a mix of terminals and non-terminals. Note that U and V must be a single terminal.
- The pair $<R_1;R_2>$ is feasible if there exists a derivation tree where R_2 is used right after R_1. This is the case if and only R_1 is reachable from the start symbol, and if V appears in Z_1.

Example





Name	Prod. rule	Feasible rule-pairs
RSB	S → Brace	<RSB;RB>
RSC	S → Curly	<RSC;RC>
RSE	S → ε	
RB	Brace → "(" S ")" S	<RB;RSB>, <RB;RSC>, <RB;RSE>
RC	Curly → "{" S "}" S	<RC;RSB>, <RC;RSC>, <RC,RSE>

These two derivations cover all rule-pairs, except <RC;RSB> and <RC;RSC>. Exercise: add few more derivations to get full pair-wise rule coverage.

Position dependent expansion

Name	Prod. rule	Feasible rule-pairs
RB	*Brace* → "(" *S* ")" *S*	<RB;RSB>, <RB;RSC>, <RB;RSE>
RSB	*S* → *Brace*	<RSB;RB>
RSC	*S* → *Curly*	<RSC;RC>
RSE	*S* → ε	

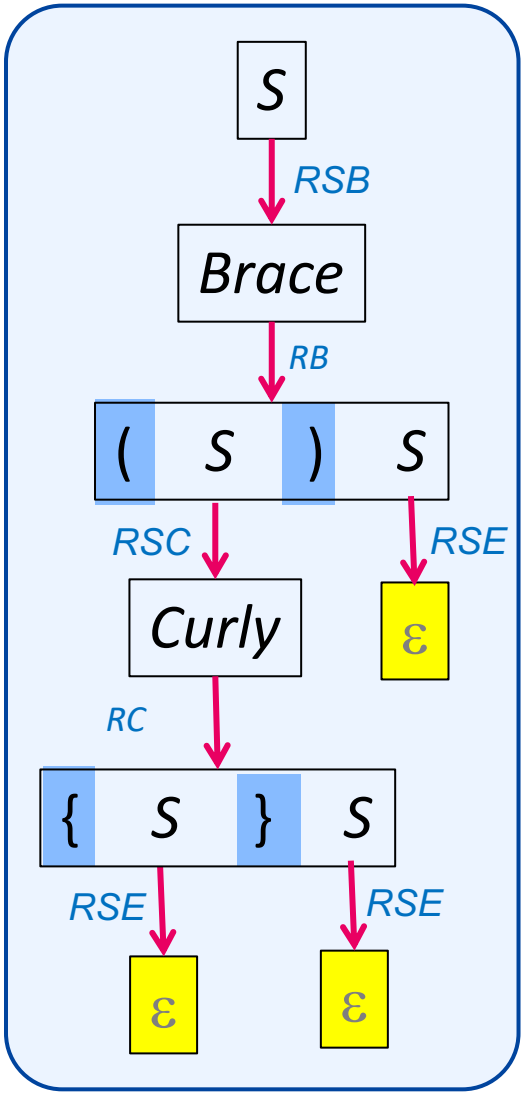
- Note that there are two non-terminals in the rule RB (two S') which can be expanded in different ways, e.g. the first S can be expanded with RSB while the second with RSC.
- Pair-wise rule coverage cannot differentiate in which position the second component of the pair is applied.

Rule-position-rule combination

- Let N be a non-terminal. Define: ***alts(N)*** = the set of N's production rules. E.g. alts(S) = { RSB, RSC, RSE }. alts(Brace) = { RB }.
- Let R_1 and R_2 be production rules of a grammar G, R_1 = A → *z* and and N is the k^{th} symbol in *z*. The tuple **<R_1;k;R_2>** is a **Rule-position-rule combination of G** if R_2 ∈ alts(N).

Name	Prod. rule	rule-pos-rule combs.
RB	*Brace* → "(" *S* ")" *S*	<RB;1;RSB> <RB;1;RSC> <RB;1;RSE> <RB;3;RSB> <RB;3;RSC> <RB;3;RSE>
RSB	*S* → *Brace*	<RSB;0;RB>
RSC	*S* → *Curly*	<RSC;0;RC>
RSE	*S* → ε	-

Covering RPR



Name	Prod. rule	RPRs
RB	*Brace* → "(" *S* ")" *S*	<RB;**1**;RSB> <RB;**1**;RSC> <RB;**1**;RSE> <RB;**3**;RSB> <RB;**3**;RSC> <RB;**3**;RSE>
RSB	*S* → *Brace*	<RSB;0;RB>
RSC	*S* → *Curly*	<RSC;0;RC>
RSE	*S* → ε	-

A derivation tree *t* covers an RPR <R_1 ; *k* ; R_2> if (1) an arrow labelled by R_1 appears in t, and (2): let V be the target node of this arrow; R_2 appears as the outgoing arrow of the k-th symbol of V.

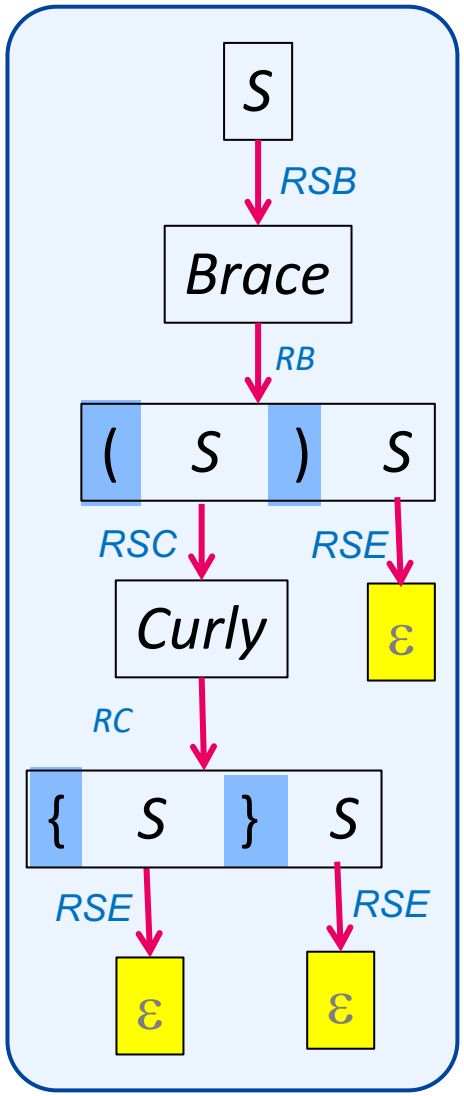
As an example, the RPRs covered by the tree to the left are marked blue (those for <RC;k;..> are not shown).

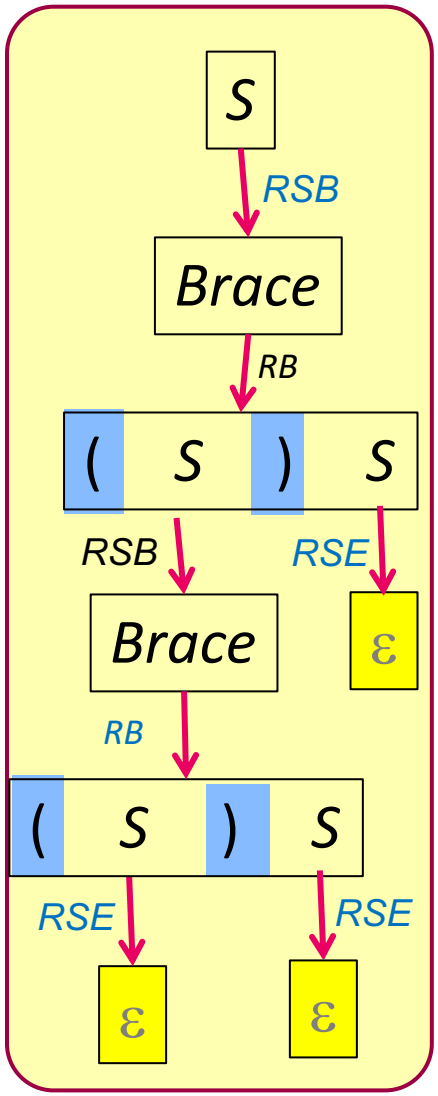
Each rule-rule coverage

Name	Prod. rule	RPRs
RB	*Brace* → "(" *S* ")" *S*	<RB;1;RSB> <RB;1;RSC> <RB;1;RSE> <RB;3;RSB> <RB;3;RSC> <RB;3;RSE>
RC	*Curly* → "{" *S* "}" *S*	<RC;1;RSB> <RC;1;RSC> <RC;1;RSE> <RC;3;RSB> <RC;3;RSC> <RC;3;RSE>
RSB	*S* → *Brace*	<RSB;0;RB>
RSC	*S* → *Curly*	<RSC;0;RC>
RSE	*S* → ε	-

Each Rule-Rule Coverage (ERRC) over a grammar G: the TR consists of all RPRs in G.

Example





Name	Prod. rule	rule-pairs	RPRs
RSB	S → Brace	<RSB;RB>	<RSB;0;RB>
RSC	S → Curly	<RSC;RC>	<RSC;0;RC>
RSE	S → ε		
RB	Brace → "(" S ")" S	<RB;RSB> <RB;RSC> <RB;RSE>	<RB;**1**;RSB> <RB;**1**;RSC> <RB;**1**;RSE> <RB;**3**;RSB> <RB;**3**;RSC> <RB;**3**;RSE>
RC	Curly → "{" S "}" S	<RC;RSB> <RC;RSC> <RC,RSE>	<RC;**1**;RSB> <RC;**1**;RSC> <RC;**1**;RSE> <RC;**3**;RSB> <RC;**3**;RSC> <RC;**3**;RSE

(blue are covered by the two test cases on the left)

Test Case size

The size of your test set (# test cases) matters, but so does the size of each test case. “Smaller” test cases are easier to understand, and if a bug is revealed, they are easier to debug.

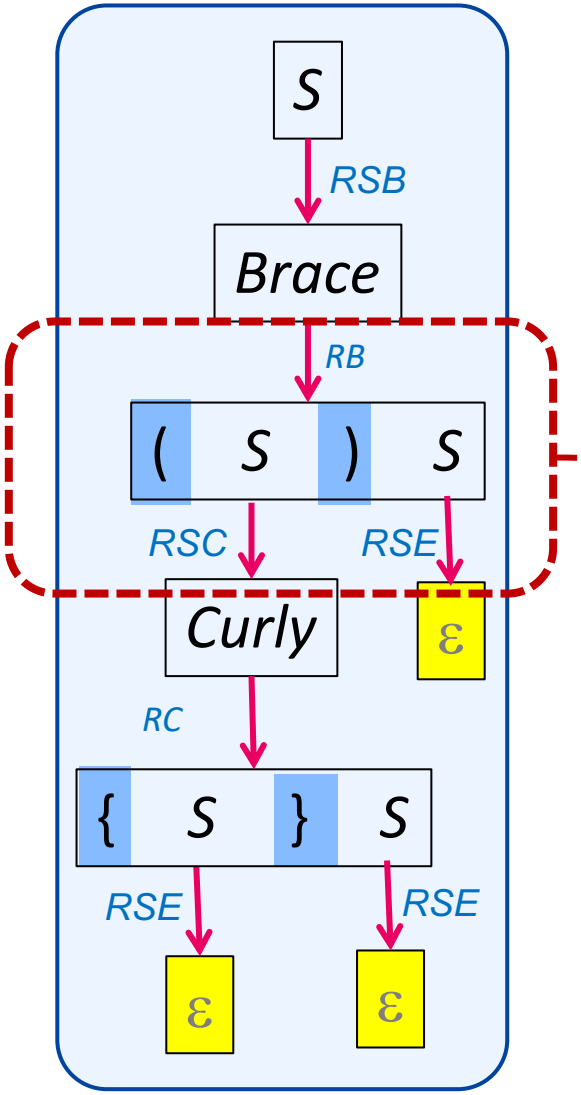
In the previous example, due to the recursion in the grammar it is actually possible to cover all rule-pairs, and even to cover all RPRs with just a single test case; but you will end up with a one relatively large and complex test case.

Rule vector combination

Let R = A → *z* be a rule of a grammar G. A vector <R;k_1;R_1, ... ,k_n;R_n> is a **rule vector combination** of G, if 0≤k_i<#*z* and R_i ∈ alts(z_{k_i}). We assume k_1... k_n to be increasing in their order.

Name	Prod. rule	rule vector combs.		
RB	*Brace* → "(" *S* ")" *S*	<RB;1;RSB;3;RSB> <RB;1;RSB;3;RSC> <RB;1;RSB;3;RSE>	<RB;1;RSC;3;RSB> <RB;1;RSC;3;RSC> <RB;1;RSC;3;RSE>	<RB;1;RSE;3;RSB> <RB;1;RSE;3;RSC> <RB;1;RSE;3;RSE>
RSB	*S* → *Brace*	<RSB;0;RB>		
RSC	*S* → *Curly*	<RSC;0;RC>		
RSE	*S* → ε	-		

Covering a rule vector



A derivation tree t **covers** a rule vector <R;k_1;R_1, ... ,k_n;R_n> if R appears as an arrow in the tree, pointing to a node V whose outgoing arrows are R_1 ... R_n (in the same order).

Example: the tree to the left covers the rule vector **<RB; 1 ; RSC ; 3 ; RSE>**

And also: **<RC; 1 ; RSE ; 3 ; RSE>**

All-rule-rule coverage

Name	Prod. rule	rule vector combs.		
RB	*Brace* → "(" *S* ")" *S*	<RB;1;RSB;3;RSB> <RB;1;RSB;3;RSC> <RB;1;RSB;3;RSE>	<RB;1;RSC;3;RSB> <RB;1;RSC;3;RSC> <RB;1;RSC;3;RSE>	<RB;1;RSE;3;RSB> <RB;1;REC;3;RSC> <RB;1;RSE;3;RSE>
RC	*Curly* → "{" *S* "}" *S*	Analogous as RB, RC also has 9 rule vector combinations.		
RSB	*S* → *Brace*	<RSB;0;RB>		
RSC	*S* → *Curly*	<RSC;0;RC>		
RSE	*S* → ε	-		

All Rule-Rule Coverage (ARRC) over a grammar G: for every rule R TR includes every rule vector combinations of R.

Subsumption