15-0: **Universal TM**

- Turing Machines are "Hard Wired"
  - Addition machine only adds
  - $0^n1^n2^n$ machine only determines if a string is in the language $0^n1^n2^n$
- Have seen one "Programmable TM"
  - Random Access Computer TM

15-1: **Universal TM**

- We can create a "Universal Turing Machine"
  - Takes as input a description of a Turing Machine, and the input string for the Turing Machine
  - Simulates running the machine on the input string
  - "Turing Machine Interpreter"
- Writing a Java Interpreter in Java, for instance, is not all that strange – essentially what we are doing with Turing Machines

15-2: **Encoding a Turing Machine**

- Our "Universal Turing Machine" needs to have a specific, finite alphabet of tape symbols
- We need to be able to simulate *any* Turing Machine with *any* tape alphabet
  - Use an encoding scheme

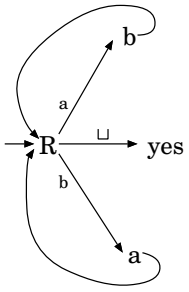
15-3: **Encoding a Turing Machine**

- Tape alphabet Σ for Universal Turing Machine:
  - $q\ a\ 0\ 1\ ,\ (\ )$
- Encoded states:
  - $q001$, $q010$, $q011$, $q100$, ...
- Encoded Tape symbols
  - $a001$, $a010$, $a011$, $a100$, ...

15-4: **Encoding a Turing Machine**

- Turing Machine that changes all a's to b's, and all b's to a's

15-5: **Encoding a Turing Machine**



	$\sqcup$	a	b
q_0	$(q_2, \sqcup)$	(q_1, b)	(q_1, a)
q_1	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$
q_3			

15-6: **Encoding a Turing Machine**

	$\sqcup$	a	b
q_0	$(q_2, \sqcup)$	(q_1, b)	(q_1, a)
q_1	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$
q_3			

Symbol	Encoding
$\sqcup$	$a000$
$\leftarrow$	$a001$
$\rightarrow$	$a010$
a	$a011$
b	$a100$

($\leftarrow$ always symbol 1, $\rightarrow$ always symbol 10_2)

$(q00, a000, q10, a000)(q00, a011, q01, a100)(q00, a100, q01, a011)\ (q01, a000, q00, a010)(q01, a011, q00, a100)(q01, a100, q00, a100)$

15-7: **Encoding a Turing Machine**

- Halting states can be coded implicitly
  - No outgoing edges = halting state
- If we want a "yes" and "no" state
  - First halting state is "yes"
  - Second halting state is "no"

15-8: **Encoding a Turing Machine**

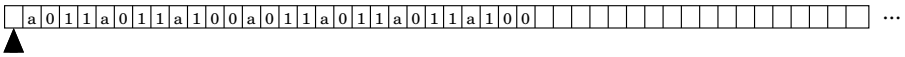
- Given any Turing Machine M, we can create an encoding of the machine, $e(M)$
- Some machines will require more "digits" to represent states & symbols
  - Why used q and a separators
  - We can actually encode any Turing Machine (and any tape) using just 0's and 1's (more on this in a minute)

15-9: **Universal Turing Machine**

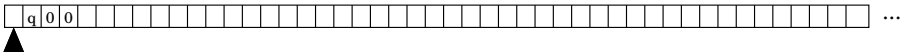
- Takes as input an encoding of a Turing Machine $e(M)$, and an encoding of the input tape $e(w)$
- Simulates running M on w
- 3-Tape Machine:
  - Simulated Tape
  - Current State Tape
  - Transition Function Tape

15-10: **Universal Turing Machine**

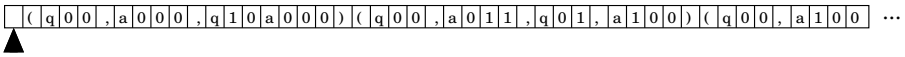
Input Tape



State Tape



Transition Function Tape



15-11: **Encoding a Turing Machine**

- Encoding a Turing Machine using just 0's and 1's:
  - If we knew how many states there were, and how many symbols in the input alphabet, we wouldn't need the separators – a q ( ) ,
  - Each encoding start with the # of digits used for states, and the # of digits used for alphabet symbols, in unary.
- $(q00, a000, q10, a000)\,(q00, a011, q01, a100)\,(q00, a100, q01, a011), (q01, a000, q00, a010)\,(q01, a011, q00, a100)$ $(q01, a100, q00, a100)$
- 1101110000001000000000011000010001011010000000100101100100011000100

15-12: **Halting Problem**

- Halting Machine takes as input an encoding of a Turing Machine $e(M)$ and an encoding of an input string $e(w)$, and returns "yes" if M halts on w, and "no" if M does not halt on w.
- Like writing a Java program that parses a Java function, and determines if that function halts on a specific input

15-13: **Language vs. Problem**
Brief Interlude

- We will use "Language" and "Problem" interchangeably
- Any Problem can be converted to a Language, and vice-versa
  - Problem: Multiply two numbers x and y
  - Language: $L = \{x; y; z : z = x * y\}$

15-14: **Language vs. Problem**
Brief Interlude

- We will use "Language" and "Problem" interchangeably
- Any Problem can be converted to a Language, and vice-versa
  - Problem: Determine if a number is prime
  - Language: $L = \{p : p \text{ is prime }\}$

15-15: **Language vs. Problem**
Brief Interlude

- We will use "Language" and "Problem" interchangeably
- Any Problem can be converted to a Language, and vice-versa
  - Problem: Determine if a Turing Machine M halts on an input string w
  - Language: $L = \{e(M), e(w) : M \text{ halts on } w\}$

15-16: **Halting Problem**

- Halting Machine takes as input an encoding of a Turing Machine $e(M)$ and an encoding of an input string $e(w)$, and returns "yes" if M halts on w, and "no" if M does not halt on w.
- Like writing a Java program that parses a Java function, and determines if that function halts on a specific input
- How might the Java version work?

15-17: **Halting Problem**

- Halting Machine takes as input an encoding of a Turing Machine $e(M)$ and an encoding of an input string $e(w)$, and returns "yes" if M halts on w, and "no" if M does not halt on w.
- Like writing a Java program that parses a Java function, and determines if that function halts on a specific input
- How might the Java version work?
  - Check for loops
  - `while (<test>) <body>`
    Use program verification techniques to see if test can ever be false, etc.

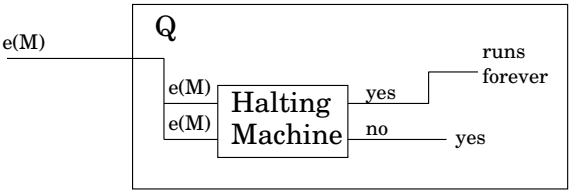
15-18: **Halting Problem**

- The Halting Problem is *Undecidable*
  - There exists no Turing Machine that decides it
  - There is no Turing Machine that halts on all inputs, and always says "yes" if M halts on w, and always says "no" if M does not halt on w
- Prove Halting Problem is Undecidable by Contradiction:

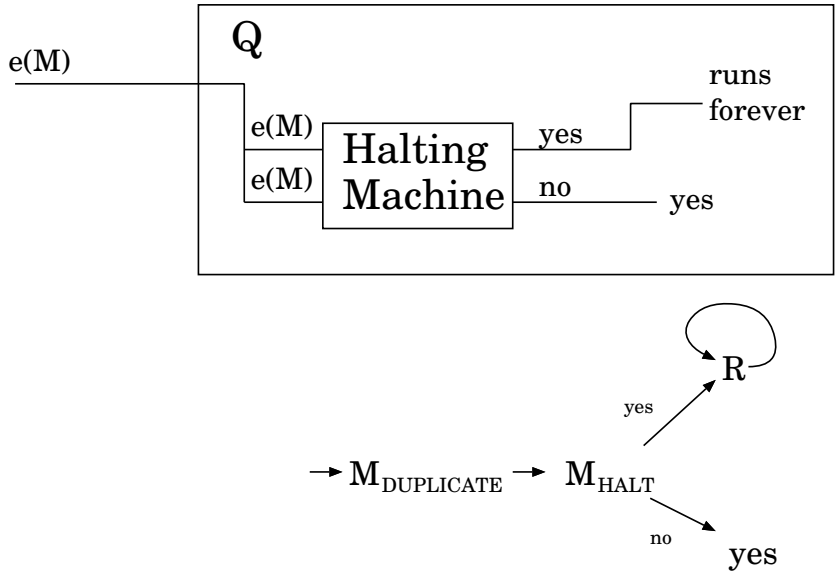
15-19: **Halting Problem**

- Prove Halting Problem is Undecidable by Contradiction:
  - Assume that there is some Turing Machine that solves the halting problem.

  - We can use this machine to create a new machine Q:



15-20: **Halting Problem**



15-21: **Halting Problem**

- Machine Q takes as input a Turing Machine M, and either halts, or runs forever.
- What happens if we run Q on $e(Q)$?
  - If M_{HALT} says Q should run forever on $e(Q)$, Q halts
  - If M_{HALT} says Q should halt on $e(Q)$, Q runs forever
- Q must not exist – but Q is easy to build if M_{HALT} exists, so M_{HALT} must not exist

15-22: **Halting Problem (Java)**

- Quick sideline: Prove that there can be no Java program that takes as input two strings, one containig source code for a Java program, and one containing an input, and determines if that program will halt when run on the given input.

```
boolean Halts(String SourceCode, String Input);
```

15-23: **Halting Problem (Java)**

```
boolean Halts(String SourceCode, String Input);

void Contrarian(String SourceCode) {
   if (Halts(SourceCode, SourceCode))
       while (true);
   else
       return;
}
```

15-24: **Halting Problem (Java)**

```
boolean Halts(String SourceCode, String Input);

void Contrarian(String SourceCode) {
```

```
   if (Halts(SourceCode, SourceCode))
       while (true);
   else
       return;
}
Contrarian("void Contrarian(String SourceCode { \
               if (Halts(SourceCode, SourceCode)) \
                ...
           } ");
```

What happens?

15-25: **Halting Problem II**

- What if we restrict the input language, to prohibit "running machine on its own encoding"?
  - Blank-Tape Halting Problem
  - Given a Turing Machine M, does M halt when run on the empty tape?

15-26: **Halting Problem II**

- What if we restrict the input language, to prohibit "running machine on its own encoding"?
  - Blank-Tape Halting Problem
  - Given a Turing Machine M, does M halt when run on the empty tape?
- This problem is *also* undecidable
- Prove using a *reduction*

15-27: **Reduction**

- Reduce Problem A to Problem B
  - Convert instance of Problem A to an instance of Problem B
    - Problem A: Power – x^y
    - Problem B: Multiplication – $x * y$
  - If we can solve Problem B, we can solve Problem A
  - If we can multiply two numbers, we can calculate the power x^y

15-28: **Reduction**

- If we can reduce Problem A to Problem B, and
- Problem A is undecidable, then:
- Problem B must also be undecidable
  - Because, if we could solve B, we could solve A

15-29: **Reduction**

- To prove a problem B is undecidable:
  - Start with a an instance of a known undecidable problem (like the Halting Problem)

- Create an instance of Problem B, such that the answer to the instance of Problem B gives the answer to the undecidable problem
- If we could solve Problem B, we could solve the halting problem . . .
- . . . thus Problem B must be undecidable

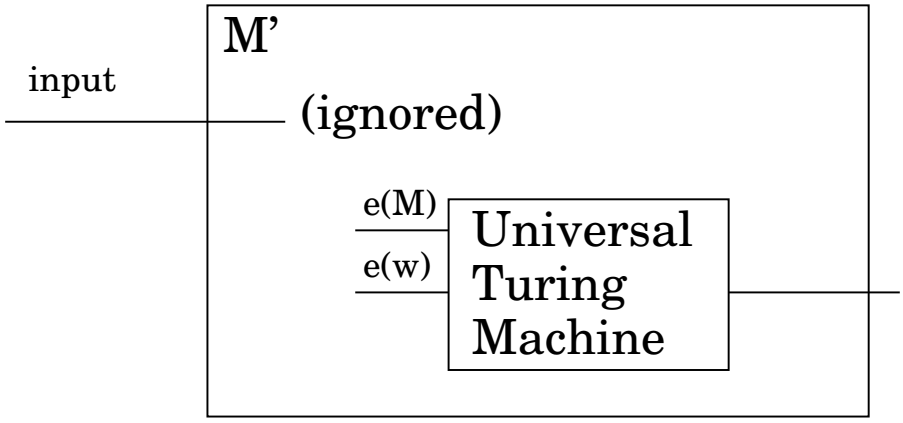
15-30: **Halting Problem II**

- Show that the Blank-Tape Halting Problem is undecidable, by reducing the Halting Problem to the Blank-Tape Halting Problem
  - Given any machine/input pair M, w, create a machine M' such that M' halts on the empty tape if and only if M halts on w

15-31: **Halting Problem II**

- Given any machine/input pair M, w, create a machine M' such that M' halts on the empty tape if and only if M halts on w
- Machine M':
  - Erase input tape (ignore input)
  - Write M, w on input tape
  - Run Universal Turing Machine
- No matter what the input to M' is, the input is ignored, and M' simulates running M on w
- M' halts on the empty tape iff M halts on w
- If we could solve the Empty-Tape Halting Problem, we could solve the standard Halting Problem

15-32: **Halting Problem II**



15-33: **Halting Problem II (Java)**

```
boolean MPrime(String Input) {
   String code = "  ... (any java source)"
   String input = "  ... (any string)"
   jvm(javac(code), input);
}
```

15-34: **More Reductions ...**

- What do we know about Problem A if we reduce it to the Halting Problem?
  - That is, given an instance of Problem A, create an instance of the halting problem, such that solution to the instance of the halting problem is the same as the solution to the instance of Problem A

15-35: **More Reductions ...**

- What do we know about Problem A if we reduce it to the Halting Problem?
  - That is, given an instance of Problem A, create an instance of the halting problem, such that solution to the instance of the halting problem is the same as the solution to the instance of Problem A
  - We know that the Halting Problem is at least as hard as Problem A
  - If we could decide the Halting Problem, we could decide Problem A

15-36: **More Reductions ...**

- What do we know about Problem A if we reduce it to the Halting Problem?
  - That is, given an instance of Problem A, create an instance of the halting problem, such that solution to the instance of the halting problem is the same as the solution to the instance of Problem A
  - We know that the Halting Problem is at least as hard as Problem A
  - If we could decide the Halting Problem, we could decide Problem A
- ... Which tells us nothing about Problem A!

15-37: **More Reductions ...**

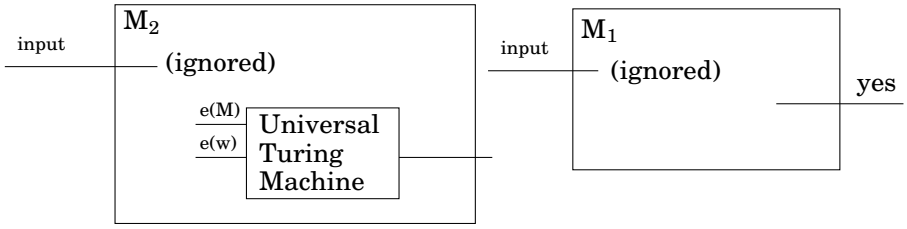
- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?

15-38: **More Reductions ...**

- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?
  - Start with an instance M, w of the halting problem
  - Create M_1, which accepts everything
  - Create M_2, which ignores its input, and runs M, w through the Universal Turing Machine. Accept if M halts on w.
- If M halts on w, then $L[M_2] = \Sigma^*$, and $L[M_1] = L[M_2]$
- If M does not halt on w, then $L[M_2] = \{\}$, and $L[M_1] \neq L[M_2]$

15-39: **More Reductions ...**

- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?



15-40: **More Reductions ...**

- If we had a machine M_{same} that took as input the encoding of two machines M_1 and M_2, and determined if $L[M_1] = L[M_2]$, we could solve the halting problem for any pair M, w:
  - Create a Machine that accepts everything (easy!). Encode this machine.
  - Create a Machine that first erases its input, then writes $e(M), e(w)$ on input, then runs Universal TM. Encode this machine
  - Feed encoded machines into M_{same}. If M_{same} says "yes", then M halts on w, otherwise M does not halt on w

15-41: **More Reductions ...**

- Is the language described by a TM M regular?
  - This problem is also undecidable
  - Use a reduction

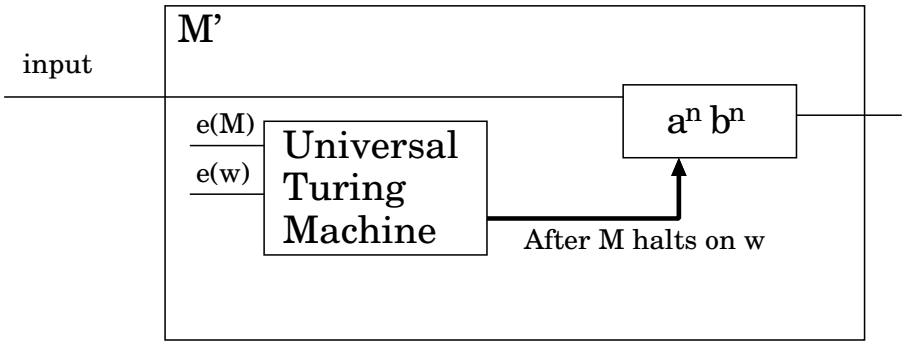
15-42: **More Reductions ...**

- Recall: to show a problem P is undecidable:
  - Pick a known undecidable problem P_{UND}
  - Create an instance of P, such that if we could solve P, we could solve P_{UND}
  - Since P_{UND} is known to be undecidable, P must be undecidable, too.

15-43: **More Reductions ...**

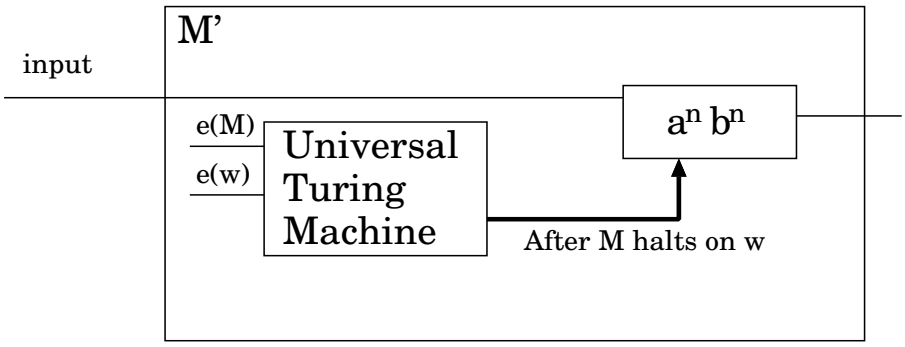
- Is the language described by a TM M regular?
  - Let M, w be an instance of the halting problem
  - Create a new machine M', that first runs M on w. If that process halts, the input string is run though a machine that accepts the language $a^n b^n$

15-44: **More Reductions ...**



- What is $L[M']$?

15-45: **More Reductions ...**



- What is $L[M']$?
  - If M halts on w, then $L[M'] = a^n b^n$, which is not regular
  - If M does not halt on w, then $L[M'] = \{\}$, which is regular

15-46: **More Reductions ...**

- So, if we have a machine M_{REG}, that took as input a Turing machine M_1, and decided if $L[M_1]$ is regular, then:
  - For *any* Turing machine M and string w, we can decide if M halts on w
  - Create M' from M and w
  - Feed M' through M_{REG}
  - If M_{REG} says "yes", then M does not halt on w. If M_{REG} says "no", then M does halt on w

15-47: **More Reductions ...**

- Given a Turing Machine M, is $|L[M]| > 0$? That is, are there *any* strings accepted by M?

15-48: **More Reductions ...**

- Given a Turing Machine M, is $|L[M]| > 0$? That is, are there *any* strings accepted by M?
  - Undecidable, by reduction from the halting problem.
  - Given any TM M and string w, we create a TM M' such that:
    - $L[M'] = \Sigma^*$ if M halts on w
    - $L[M'] = \{\}$ otherwise

15-49: **More Reductions ...**

- Given a Turing Machine M, is $|L[M]| > 0$? That is, are there *any* strings accepted by M?
- Consider M':
  - Erases input
  - Simulates running M on w
  - Accepts

15-50: **Questions about Grammars**

- The following questions about unrestricted grammars are all undecidable:

- Given a Grammar G and string w, is $w \in L[G]$?
- Given a Grammar G, is $\epsilon \in L[G]$?
- Given Grammars G_1 and G_2, is $L[G_1] = L[G_2]$?
- Given a Grammar G, is $L[G] = \{\}$

15-51: **Questions about Grammars**

- The following questions about unrestricted grammars are all undecidable:
  - Given a Grammar G and string w, is $w \in L[G]$?
  - By reduction from the Halting Problem:
    - Given any Machine M, we can construct an unrestricted Grammar G, such that $L[G] = L[M]$
    - $w \in L[M]$ iff $w \in L[G]$

15-52: **Questions about Grammars**

- The following questions about Context-Free Grammars *are* decidable:
  - Given a Grammar G and string w, is $w \in L[G]$?
    - Compilers would be hard to write, otherwise
  - Given a Grammar G, is $\epsilon \in L[G]$?
    - This is a special case of determining if $w \in L[G]$

15-53: **Questions about Grammars**

- However, there are some problems about CFGs that are *not* decidable:
  - Given any CFG G, is $L[G] = \Sigma^*$
  - Given any two CFGs G_1 and G_2, is $L[G_1] = L[G_2]$
  - Given two PDA M_1 and M_2, is $L[M_1] = L[M_2]$
  - Given a PDA M, find an equivalent PDA with the smallest possible number of states

15-54: **Questions about Grammars**

- Given any CFG G, is $L[G] = \Sigma^*$?
  - Prove this problem is undecidable by reduction from the problem "Given an unrestricted grammar G, is $L[G] = \{\}$
  - That is, given any unrestricted grammar G, we will create a CFG G':
    - $L[G'] = \Sigma^*$ iff $L[G] = \{\}$

15-55: **Questions about Grammars**

- Given any unrestricted grammar G, we will create a CFG G', such that $L[G'] = \Sigma^*$ iff $L[G] = \{\}$
- First, we will modify G, to create an equivalent grammar

$S \to aBSc$
$S \to X$
$Ba \to aB$
$BX \to Xb$
$aX \to a$

15-56: **Questions about Grammars**

- Given any unrestricted grammar G, we will create a CFG G', such that $L[G'] = \Sigma^*$ iff $L[G] = \{\}$
- First, we will modify G, to create an equivalent grammar

$S \to A_1$	$A_1 \to aBSc$
$S \to A_2$	$A_2 \to X$
$Ba \to A_3$	$A_3 \to aB$
$BX \to A_4$	$A_4 \to Xb$
$aX \to A_5$	$A_5 \to a$

15-57: **Questions about Grammars**

- A **standard derivation** in this new grammar is one in which each odd step applies of rule of the form $u_i \to A_i$, and every even step applies a rule of the form $A_i \to v_i$

$S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow aBA_2c \Rightarrow aBXc$
$\Rightarrow aA_4c \Rightarrow aXbc \Rightarrow A_5bc \Rightarrow abc$

$S \to A_1$	$A_1 \to aBSc$
$S \to A_2$	$A_2 \to X$
$Ba \to A_3$	$A_3 \to aB$
$BX \to A_4$	$A_4 \to Xb$
$aX \to A_5$	$A_5 \to a$

15-58: **Questions about Grammars**

- Each standard derivation of a string generated from G can be considered a string over the alphabet $V \cup \{\Rightarrow\}$ (recall V contains Σ as well as non-terminals)
- We can define a new langauge D_G, the set of all valid standard derivations of string generated by G.

"$S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow aBA_2c \Rightarrow aBXc \Rightarrow aA_4c \Rightarrow aXbc \Rightarrow A_5bc \Rightarrow abc$" $\in D_G$

"$S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow aBA_1c \Rightarrow aBaBScc \Rightarrow aBaBA_2cc \Rightarrow aBaBXcc \Rightarrow aA_3BXcc \Rightarrow aaBBXcc \Rightarrow aaBA_4cc \Rightarrow aaBXbcc \Rightarrow aaA_4bcc \Rightarrow aaXbbcc \Rightarrow aA_5bbcc \Rightarrow aabbcc$" $\in D_G$

15-59: **Questions about Grammars**

- A **boustrophedon version** of a derivation is one in which the odd numbered elements of the derivation are reversed:

$$S \Rightarrow x_1^R \Rightarrow x_2 \Rightarrow x_3^R \Rightarrow \ldots \Rightarrow x_{n-1}^R \Rightarrow x_n$$

- Derivation

  $S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow aBA_2c \Rightarrow aBXc$
  $\Rightarrow aA_4c \Rightarrow aXbc \Rightarrow A_5bc \Rightarrow abc$

- Boustrophedon version of the derivation

  $S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow cA_2Ba \Rightarrow aBXc$
  $\Rightarrow cA_4a \Rightarrow aXbc \Rightarrow cbA_5 \Rightarrow abc$

15-60: **Questions about Grammars**

- D_G is the language of all standard derivations of strings in $L[G]$
- BD_G is the language of all boustrophedon versions of standard derivations of strings in $L[G]$

"$S \Rightarrow A_1 \Rightarrow aBSc \Rightarrow cA_2Ba \Rightarrow aBXc \Rightarrow cA_4a \Rightarrow aXbc \Rightarrow cbA_5 \Rightarrow abc$" $\in BD_G$

15-61: **Questions about Grammars**

- Given any Unrestricted Grammar G:
  - $L[G]$ is the set of all strings generated by G
  - D_G is the set of all strings that represent standard derivations of strings generated by G
  - BD_G is the set of all strings that represent boustrophedon versions of standard derivations of strings in $L[G]$
  - $\overline{BD_G}$ is the set of all strings that do *not* represent boustrophedon versions of standard derivations of strings in $L[G]$.
    - $w \in \overline{BD_G}$ if w represents only a partial derivation, or an incorrect derivation, or a mal-formed derivation

15-62: **Questions about Grammars**

- What does it mean if $\overline{BD_G} = \Sigma^*$?

15-63: **Questions about Grammars**

- What does it mean if $\overline{BD_G} = \Sigma^*$?
  - $BD_G = \{\}$
  - $D_G = \{\}$
  - $L[G] = \{\}$
- So, if we could build a CFG that generates $\overline{BD_G}$, for any unrestricted grammar G, and we could determine if $L[G'] = \Sigma^*$ for any CFG G' ...

15-64: **Questions about Grammars**

- What does it mean if $\overline{BD_G} = \Sigma^*$?
  - $BD_G = \{\}$
  - $D_G = \{\}$
  - $L[G] = \{\}$
- So, if we could build a CFG that generates $\overline{BD_G}$, for any unrestricted grammar G, and we could determine if $L[G'] = \Sigma^*$ for any CFG G' ...
- We could determine if $L[G] = \{\}$ for any unrestricted grammar G – which means we could solve the halting problem (why?)

15-65: **Building a CFG for $\overline{BD_G}$**

- When is $w \notin BD_G$?
  - w does not start with $S \Rightarrow$
  - w does not end with $\Rightarrow v$, $v \in \Sigma^*$
  - w contains an odd # of $\Rightarrow$'s
  - w is of the form $u \Rightarrow y \Rightarrow v$, or $u \Rightarrow y$

- u contains an even number of $\Rightarrow$'s
- y contains exactly one $\Rightarrow$
- y is *not* of the form $y = y_1 A_i y_2 \Rightarrow y_2^R \beta_i y_1^R$ for some $i \leq |R|$, $y_1, y_2 \in V^*$, where β_i is the right-hand side of the ith rule in G

. . . (there's more)

15-66: **Building a CFG for $\overline{BD_G}$**

- When is $w \notin BD_G$?
  - . . .
  - w is of the form $u \Rightarrow y \Rightarrow v$
    - u contains an odd number of $\Rightarrow$'s
    - y contains exactly one $\Rightarrow$
    - y is *not* of the form $y = y_1 \alpha_i y_2 \Rightarrow y_2^R A_i y_1^R$ for some $i \leq |R|$, $y_1, y_2 \in V^*$, where α_i is the right-hand side of the ith rule in G

15-67: **Building a CFG for $\overline{BD_G}$**

- w does not start with $S \Rightarrow$
  - We can create a CFG for all strings $w \in (V \cup \Rightarrow)^*$ that do *not* start with $S \Rightarrow$
  - This language is regular, we could even creatre a DFA or regular expression for it.

15-68: **Building a CFG for $\overline{BD_G}$**

- w does not end with $\Rightarrow v$, $v \in \Sigma^*$
  - We can create a CFG for all strings $w \in (V \cup \Rightarrow)^*$ that do *not* end with $\Rightarrow v$, $v \in \Sigma^*$
  - This language is regular, we could even create a DFA or regular expression for it.

15-69: **Building a CFG for $\overline{BD_G}$**

- w does contains an odd # of $\Rightarrow$s
  - We can create a CFG for all strings $w \in (V \cup \Rightarrow)^*$ that contain an odd # of $\Rightarrow$'s
  - This language is regular, we could even create a DFA or regular expression for it.

15-70: **Building a CFG for $\overline{BD_G}$**

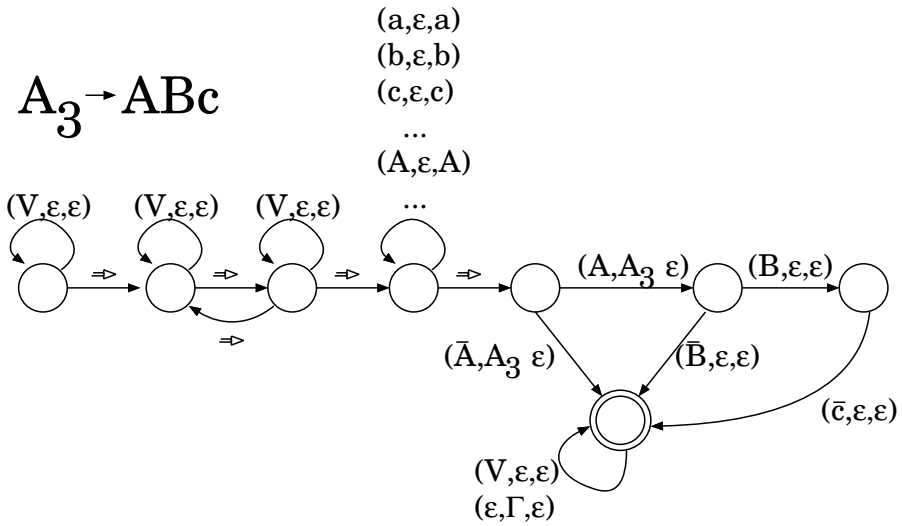
- w is of the form $u \Rightarrow y \Rightarrow v$, or $u \Rightarrow y$
  - u contains an even number of $\Rightarrow$'s
  - y contains exactly one $\Rightarrow$
  - y is *not* of the form $y = y_1 A_i y_2 \Rightarrow y_2^R \beta_i y_1^R$ for some $i \leq |R|$, $y_1, y_2 \in V^*$, where β_i is the right-hand side of the ith rule in G
- We can create a PDA which accepts strings w of this form

15-71: **Building a CFG for $\overline{BD_G}$**

- We can create a PDA which can accept all strings w of this form

- First, check that the first part of the string is of the form: $V^*(\Rightarrow V^* \Rightarrow V^*)^*$
  - Non-deterministically decide when to stop
- Push Symbols on stack until a $\Rightarrow$
- Check the input against the stack, making sure there is at least one mismatch
  - Just like the PDA for non-palindromes

15-72: **Building a CFG for $\overline{BD_G}$**



15-73: **Building a CFG for $\overline{BD_G}$**

- w is of the form $u \Rightarrow y \Rightarrow v$
  - u contains an odd number of $\Rightarrow$'s
  - y contains exactly one $\Rightarrow$
  - y is *not* of the form $y = y_1\alpha_i y_2 \Rightarrow y_2^R A_i y_1^R$ for some $i \leq |R|$, $y_1, y_2 \in V^*$, where α_i is the right-hand side of the ith rule in G
- We can create a PDA which accepts all strings w of this form.

15-74: **Questions about Grammars**

- If we could determine, for any CFG G, if $L[G] = \Sigma^*$, we could solve the halting problem
- Given any TM M and string w:
  - Create a new TM M', that accepts Σ^* if M halts on w, and $\{\}$ otherwise
  - Create an Unrestricted Grammar, such that $L[G] = L[M']$
  - Create a CFG G' for $\overline{BD_G}$
  - $L[G'] = \Sigma^*$ if and only if $L[G] = \{\}$, $L[M'] = \{\}$, and M does not halt on w

15-75: **Questions about Grammars**

- Undecidable problems about CFGs
  - Given any CFG G, is $L[G] = \Sigma^*$

    - Just proved
  - Given any two CFGs G_1 and G_2, is $L[G_1] = L[G_2]$
  - Given two PDA M_1 and M_2, is $L[M_1] = L[M_2]$
  - Given a PDA M, find an equivalent PDA with the smallest possible number of states

15-76: **Questions about Grammars**

- Given any two CFGs G_1 and G_2, is $L[G_1] = L[G_2]$
  - We can easily create a CFG G_2 which generates Σ^*
    - How?
    - Note that the preceding proof did not say "we cannot decide if $L[G] = \Sigma^*$ for any grammar G," but instead, "we cannot decide if $L[G] = \Sigma^*$ for *every* grammar G.
  - For any CFG G_1, $L[G_1] = L[G_2]$ if and only if $L[G_1] = \Sigma^*$

15-77: **Questions about Grammars**

- Given two PDA M_1 and M_2, is $L[M_1] = L[M_2]$
  - We can convert a CFG to an equivalent PDA
  - If we could determine if two PDA are equivalent:
    - We could determine if two CFGs are equivalent
    - We could determine if a CFG accepted Σ^*
    - We could determine if an Unrestricted Grammar generated any strings
    - We could determine if a Turing Machine accepted any strings
    - We could solve the halting problem

15-78: **Questions about PDA**

- Given a PDA M, find an equivalent PDA with the smallest possible number of states
  - First, prove that it is decidable whether a PDA *with one state* accepts Σ^*

15-79: **Questions about PDA**

- It is decidable whether a PDA with a single state accepts Σ^*
  - A PDA that accepts Σ^* must accept Σ.
  - We can test whether a PDA accepts Σ
    - Is $w \in L[M]$ for a PDA M is decidable
    - Test each of the $|\Sigma|$ strings sequentially
  - We can decide if a PDA M accepts Σ

15-80: **Questions about PDA**

- If a *Single State* PDA M accepts Σ, then M accepts Σ^*.
  - Why?

15-81: **Questions about PDA**

- If a *Single State* PDA M accepts Σ, then M accepts Σ^*.
  - Start in the initial state (which is also a final state) with an empty stack
  - After reading a single character, stack is empty, and we're (still!) in the initial (and final!) state – accept the string
  - We are in *exactly the same position* after reading one symbol as we were before reading in anything
  - After reading the next symbol, we will be in a final state with an empty stack – accept the string

15-82: **Questions about PDA**

- Given a PDA M, find an equivalent PDA with the smallest possible number of states
  - It is decidable whether a PDA *with one state* accepts Σ^*
  - So ...

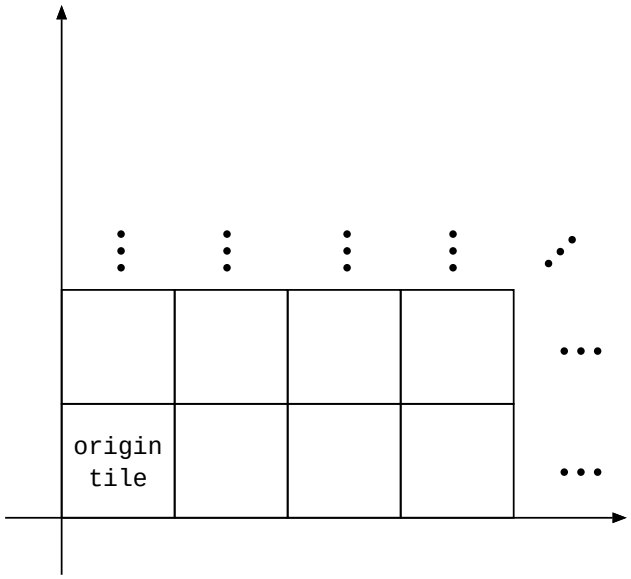
15-83: **Questions about PDA**

- Given a PDA M, find an equivalent PDA with the smallest possible number of states
  - It is decidable whether a PDA *with one state* accepts Σ^*
  - If we could minimize the number of states in a PDA, we could decide if a PDA accepted Σ^*
    - Minimize the number of states.
    - PDA accepts Σ^* iff minimized PDA has a single state, and minimized PDA accepts Σ

15-84: **Tiling Question**

- There are some problems which seem to have no relation to Turing Machines at all, which turn out to be undecidable
- Tiling Problem:
  - Set of tiles (infinite # of copies of each tile)
  - Rules for which tiles can be placed next to which other tiles (think puzzle pieces)
  - Can we tile the entire plane?

15-85: **Tiling Question**



15-86: **Tiling Question**

- Tiling System $\mathbf{D} = (D, d_0, H, V)$
  - D is a set of tiles
  - $d \in D$ is the origin tile
  - $H \subset D \times D$ list of pairs of which tiles can be next to each other
  - $V \subset D \times D$ list of pairs of which tiles can be on top of each other

15-87: **Tiling Question**

- Tiling function $f : N \times N \mapsto D$
  - Specifies which tile goes where
  - $f(0, 0) = d_0$
  - $(f(m, n), f(m + 1, n)) \in H$
  - $(f(m, n), f(m, n + 1)) \in V$

15-88: **Tiling Question**

- Problem:
  - Given a tiling system $\mathbf{D} = (D, d_0, H, V)$, does a tiling exists?
  - That is, is there a completely defined tiling function f that satisfies the requirements:
    - $f(0, 0) = d_0$
    - $(f(m, n), f(m + 1, n)) \in H$
    - $(f(m, n), f(m, n + 1)) \in V$
- This problem is undecidable!

15-89: **Tiling Question**

- Tiling Problem is undecidable
- Proof by reduction from the empty tape halting problem
  - Given any Turing Machine M
  - Create a tiling system $\mathbf{D}$
  - A tiling will exist for $\mathbf{D}$ if and only if M does not halt when run on the empty tape

15-90: **Tiling Question**

- Given any Turing Machine M, create a tiling system $\mathbf{D}$
- Basic Idea:
  - Each row of the tiling represents a TM configuration
    - Infinite row, infinite tape
  - Create rules so that successive rows i and j are only legal if TM can transition form configuration i to configuration j in one step
  - Can tile the entire plane if and only if TM does not halt

15-91: **Tiling Question**

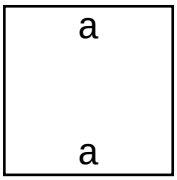
- Label the edges of each tile
- Two tiles can only be adjacent if the edges match
  - Just like a standard jigsaw puzzle

15-92: **Tiling Question**

- We will modify the Turing Machine M slightly to get M' (makes creating the tiling system a little easier)
  - Add a new symbol $>$ to the tape symbols of M
  - Add a new start state s'
  - Add a transition $((s', >), (s, \rightarrow))$
- M halts on empty tape if and only if M' halts on the tape containing $>$

15-93: **Tiling Question**

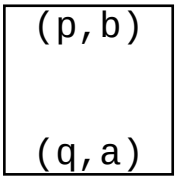
- Tiles:
  - For each symbol a that can appear on the tape of Turing Machine M, add the tile:



  - Top and bottom edges labeled with a, left and right edges labeled with ϵ

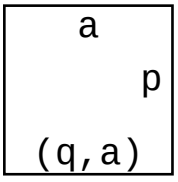
15-94: **Tiling Question**

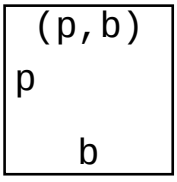
- Tiles:
  - For each transition $((q, a), (p, b))$ in δ_M add the tile:



15-95: **Tiling Question**

- Tiles:
  - For each transition $((q, a), (p, \rightarrow))$ in δ_M add the tiles:

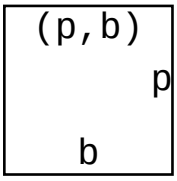


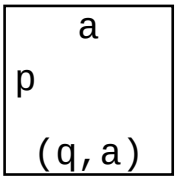


  - Add a copy of right-hand tile for every symbol b

15-96: **Tiling Question**

- Tiles:
  - For each transition $((q, a), (p, \leftarrow))$ in δ_M add the tiles:

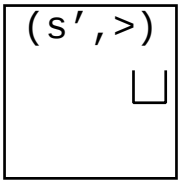




  - Add a copy of left-hand tile for every symbol b

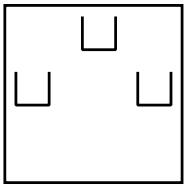
15-97: **Tiling Question**

- Initial Tile:



15-98: **Tiling Question**

- One final tile:



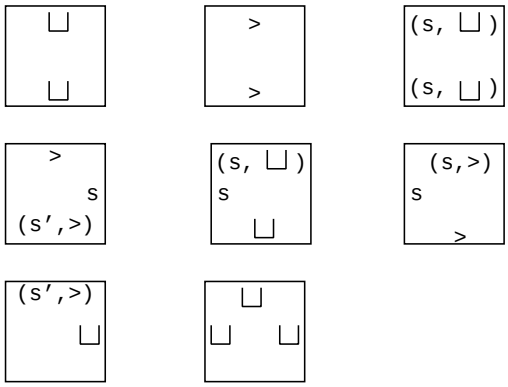
15-99: **Tiling Question**

- This tiling system has a tiling if and only if M does not halt on the empty tape.
- Example:

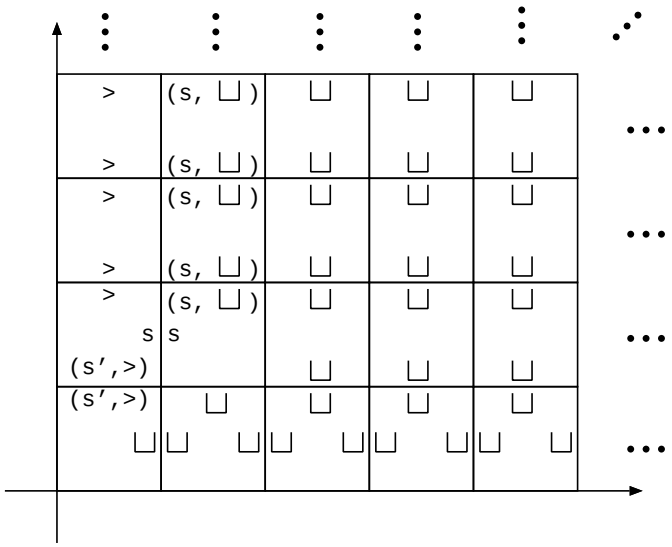
	$>$	$\sqcup$
s'	$(s, \rightarrow)$	
s		$(s, \sqcup)$

15-100: **Tiling Question**

	$>$	$\sqcup$
s'	$(s, \rightarrow)$	
s		$(s, \sqcup)$



15-101: **Tiling Question**

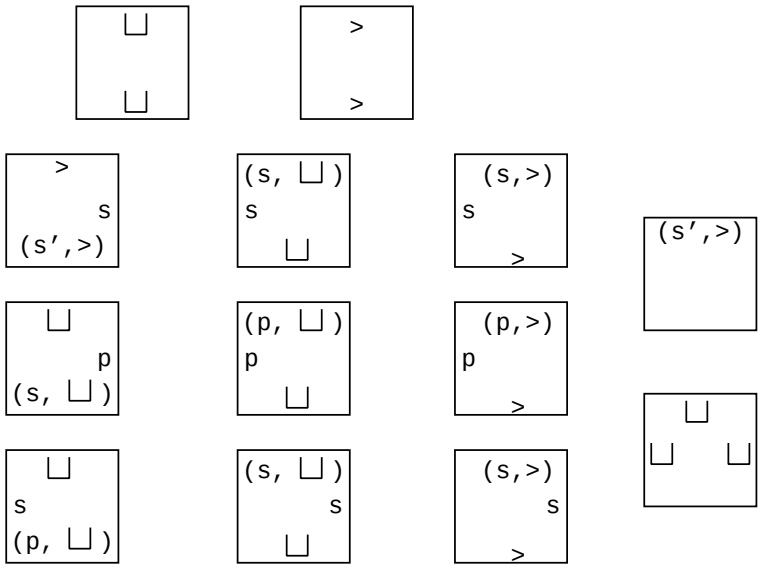


15-102: **Tiling Question**

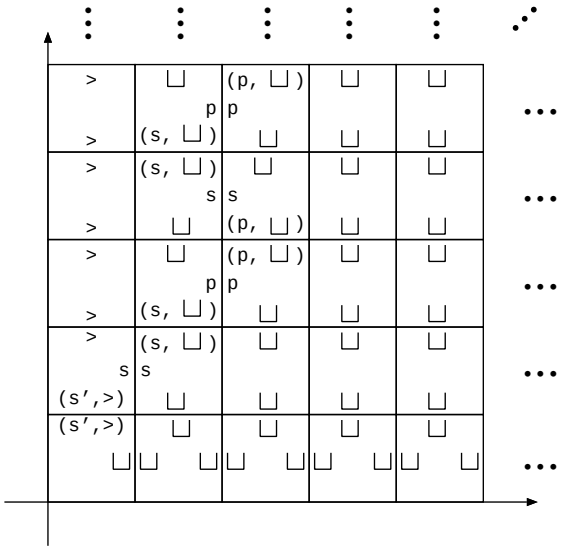
- Example II:

	$>$	$\sqcup$
s'	$(s, \rightarrow)$	
s		$(p, \rightarrow)$
p		$(s, \leftarrow)$

15-103: **Tiling Question**



15-104: **Tiling Question**

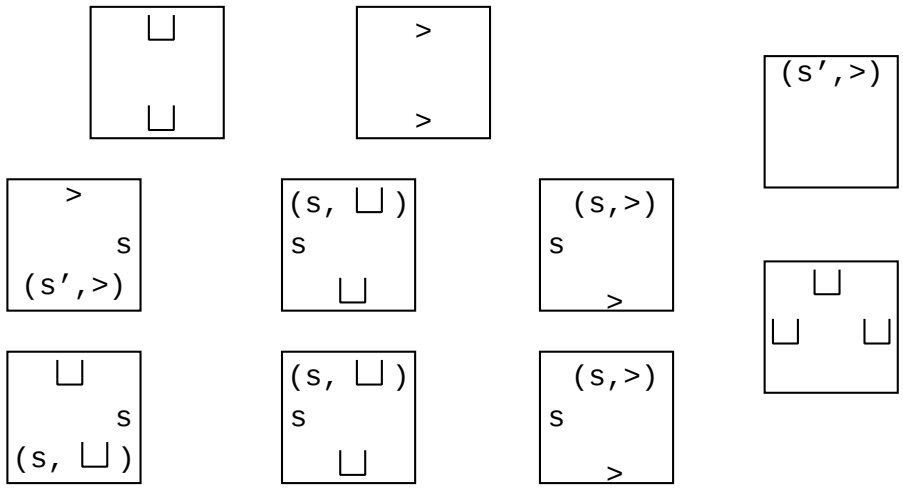


15-105: **Tiling Question**

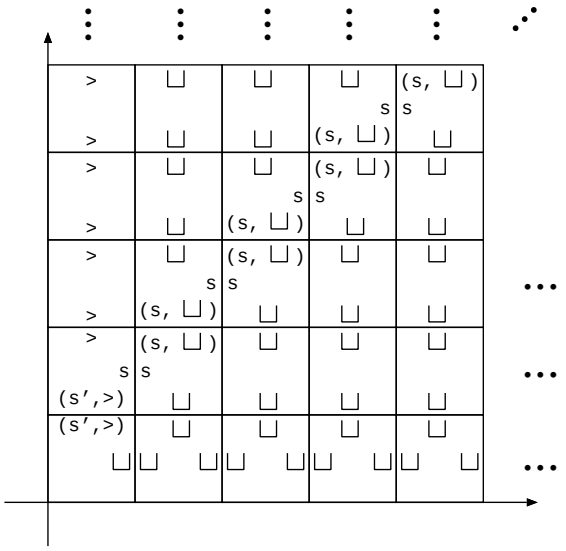
- Example III:

	$>$	$\sqcup$
s'	$(s, \rightarrow)$	
s		$(s, \rightarrow)$

15-106: **Tiling Question**



15-107: **Tiling Question**

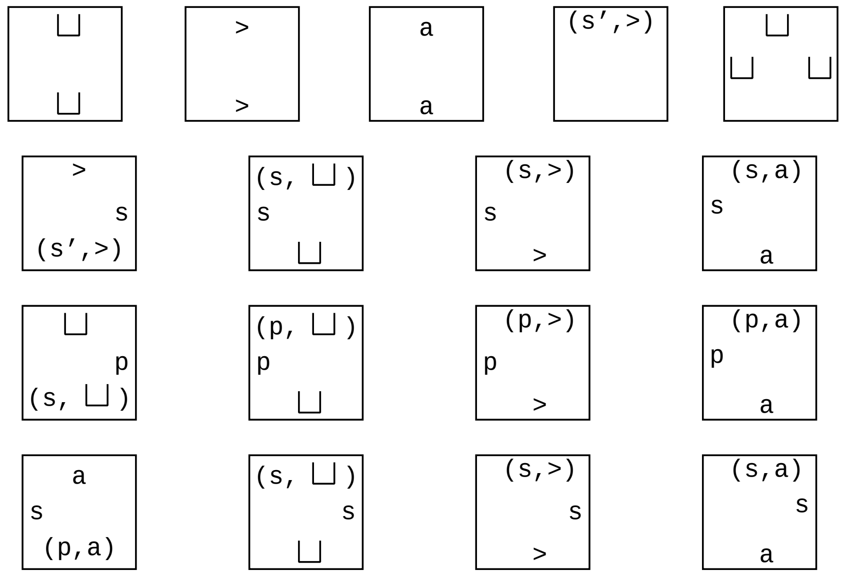


15-108: **Tiling Question**

- Example IV:

	$>$	$\sqcup$	a
s'	$(s, \rightarrow)$		
s		$(p, \rightarrow)$	
p			$(s, \leftarrow)$

15-109: **Tiling Question**



15-110: **Tiling Question**