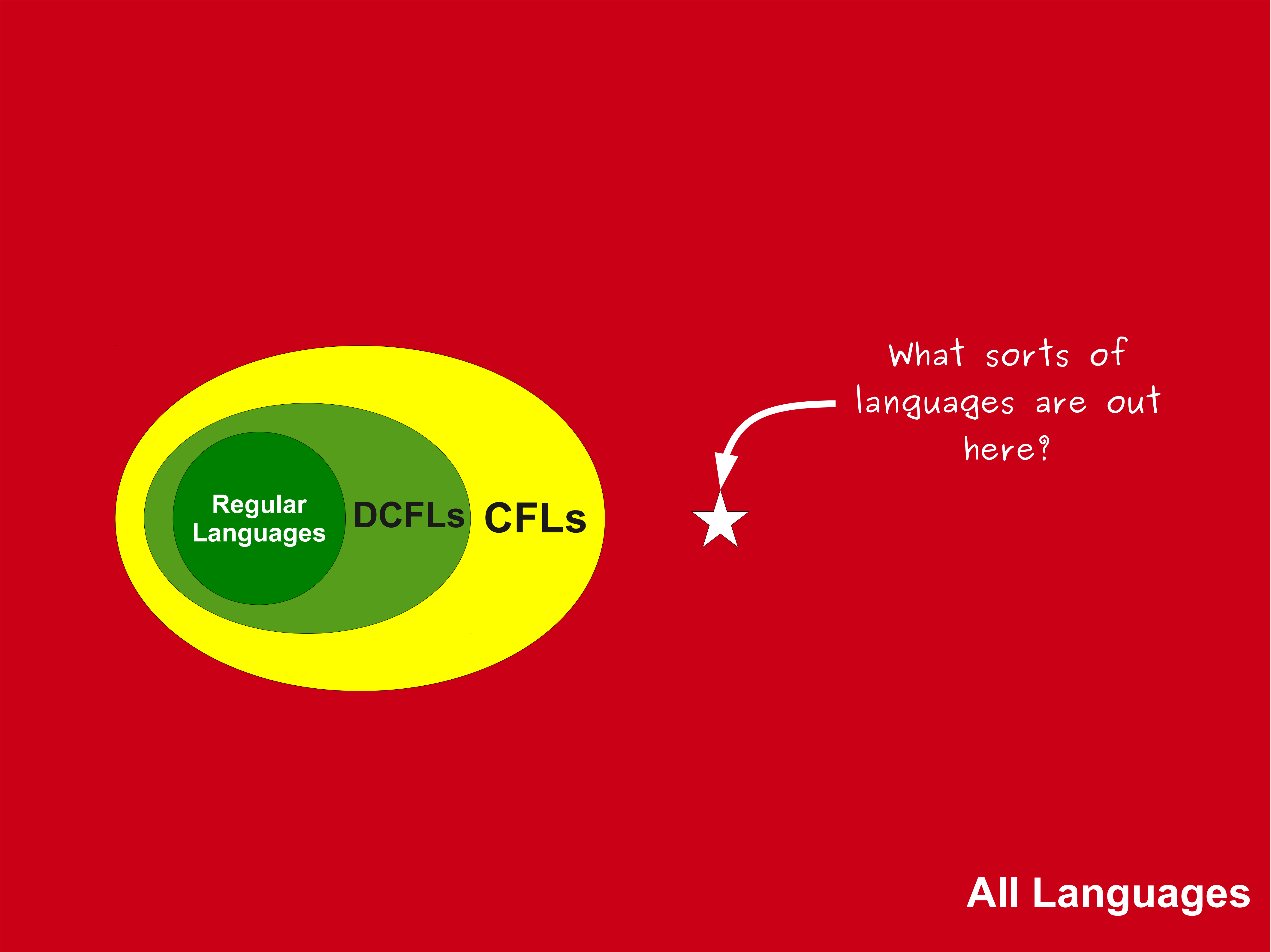
Beyond Context-Free Languages

Are some problems inherently harder than others?


Regular Languages
DCFLs
CFLs
What sorts of languages are out here?
All Languages

The Pumping Lemma for Regular Languages

- Let L be a regular language, so there is a DFA D for L.
- A sufficiently long string $w \in L$ must visit some state in D twice.
- This means w went through a loop in the D.
- By replicating the characters that went through the loop in the D, we can "pump" a portion of w to produce new strings in the language.

The Pumping Lemma Intuition

- The model of computation used has a finite description.
- For sufficiently long strings, the model of computation must repeat some step of the computation to recognize the string.
- Under the right circumstances, we can iterate this repeated step zero or more times to produce more and more strings.

Recall: Parse Trees

$\mathbf{R} \rightarrow$ `a` | `b` | `c` | ...

$\mathbf{R} \rightarrow$ "`ε`"

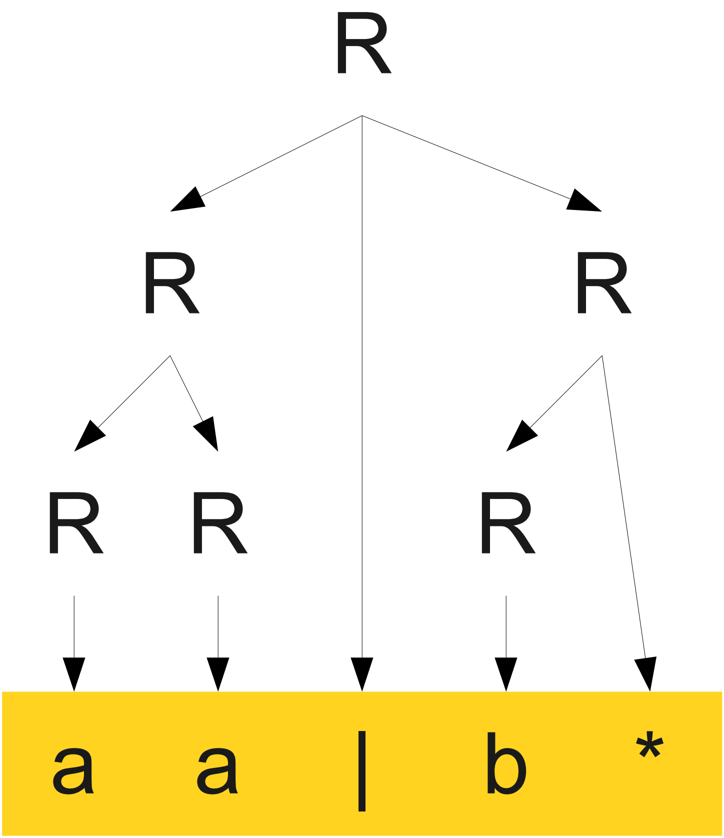
$\mathbf{R} \rightarrow$ `Ø`

$\mathbf{R} \rightarrow \mathbf{RR}$

$\mathbf{R} \rightarrow \mathbf{R}$ "`|`" $\mathbf{R}$

$\mathbf{R} \rightarrow \mathbf{R}$`*`

$\mathbf{R} \rightarrow$ `(`$\mathbf{R}$`)`

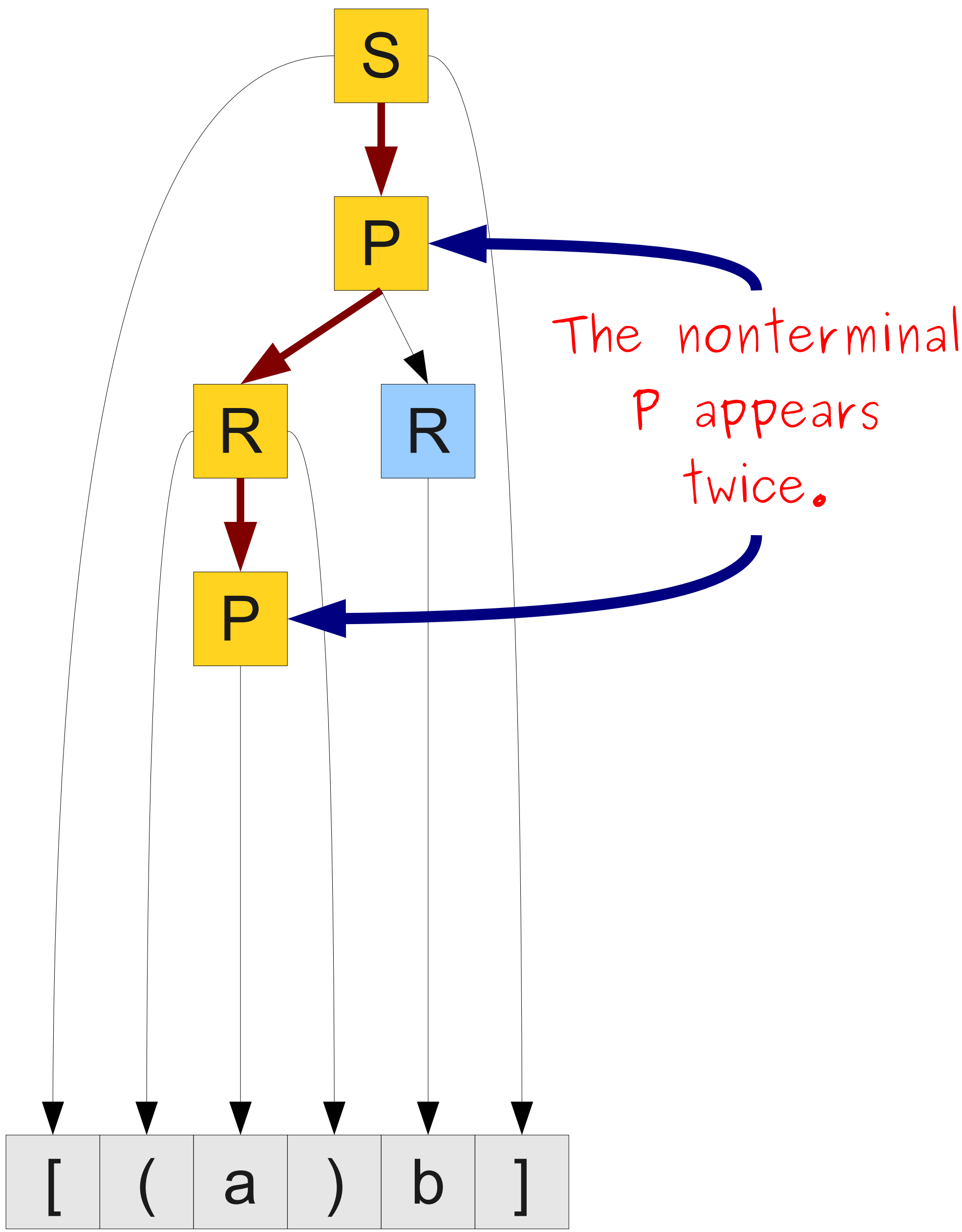


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

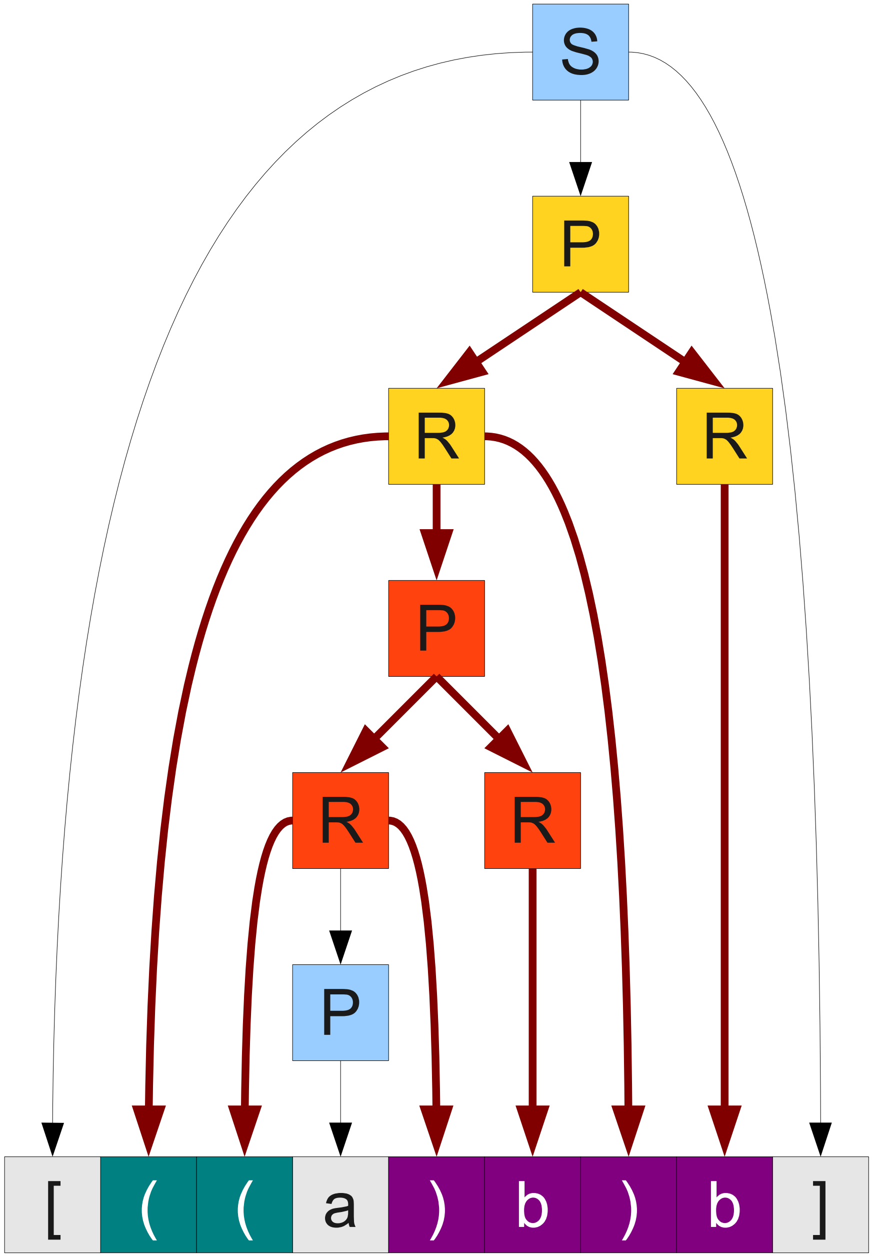


Parse Trees Revisited

$\mathbf{S} \rightarrow$ `[`$\mathbf{P}$`]`

$\mathbf{P} \rightarrow \mathbf{RR}$ | `a`

$\mathbf{R} \rightarrow$ `(`$\mathbf{P}$`)` | `b`

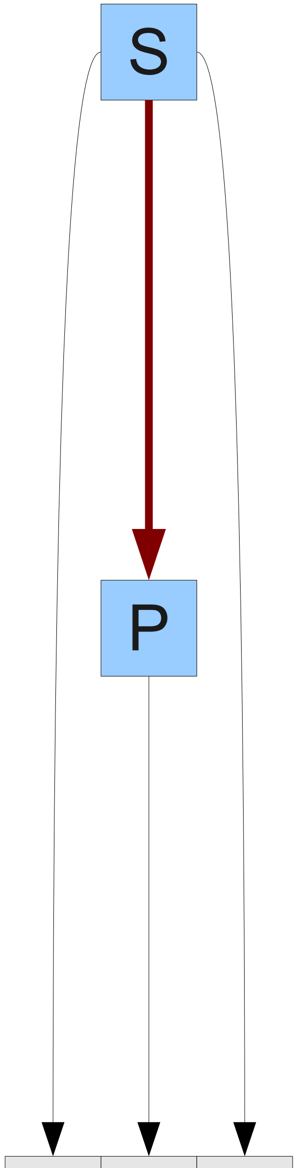


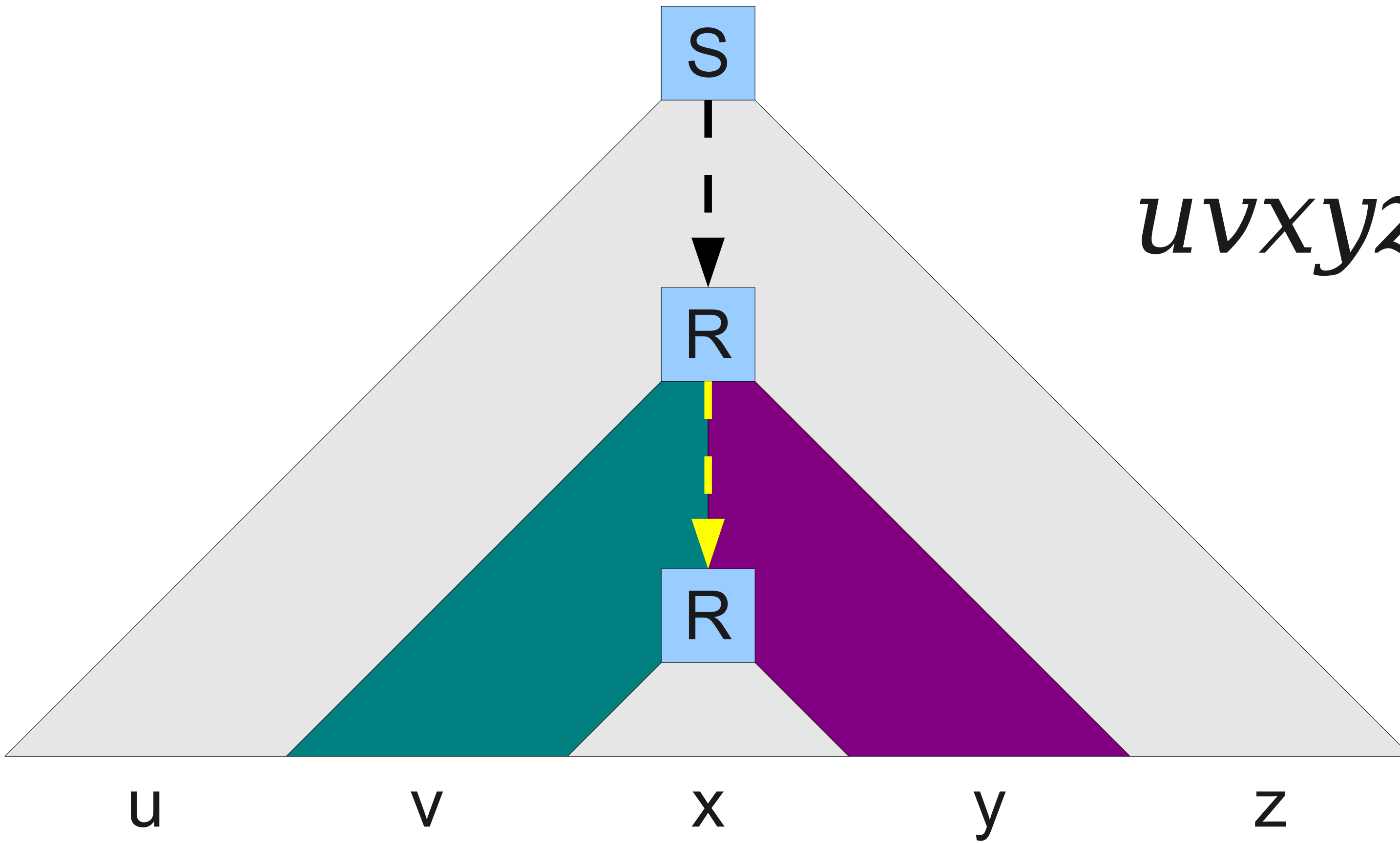
Parse Trees Revisited

S → `[`**P**`]`

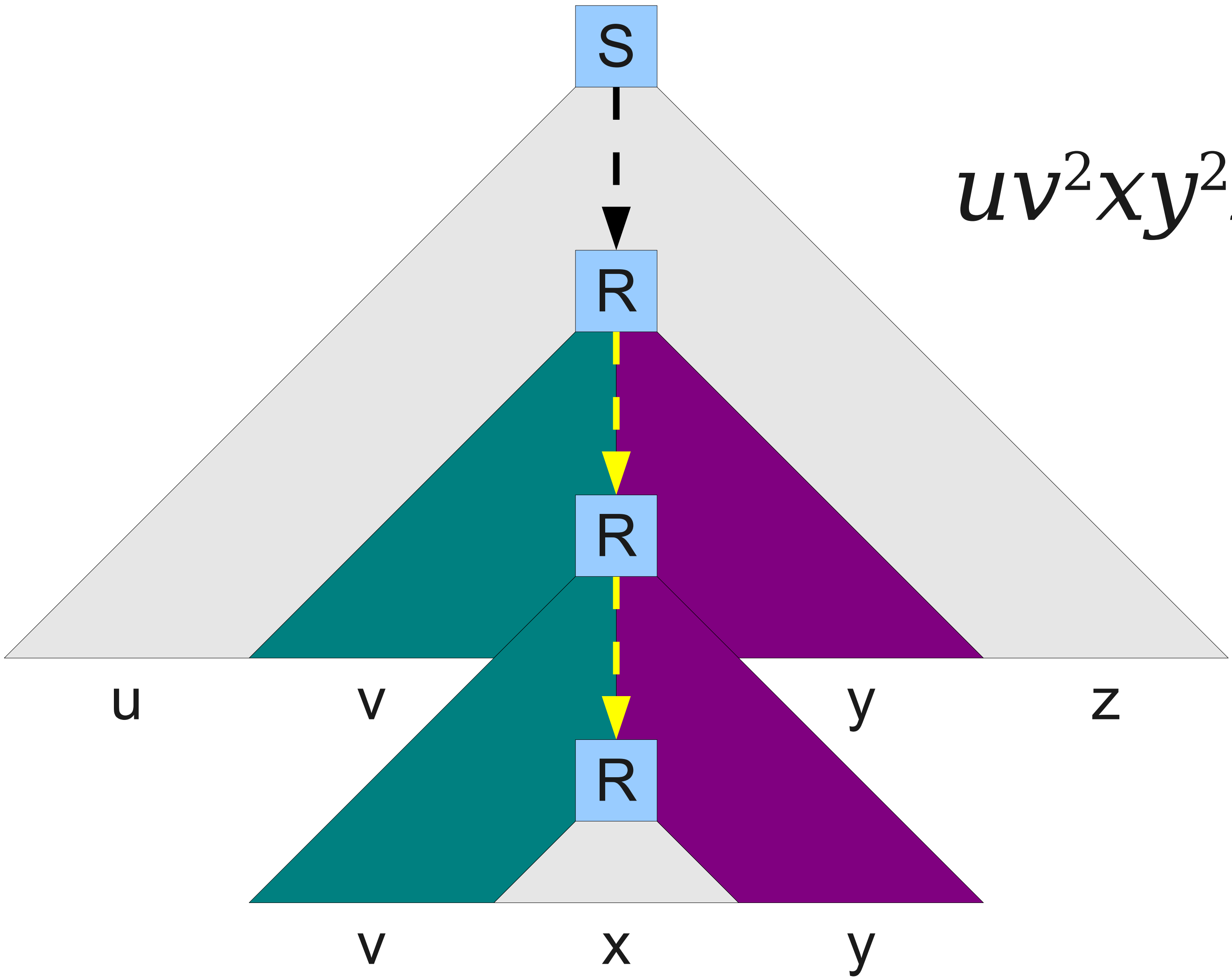
P → **RR** | `a`

R → `(`**P**`)` | `b`

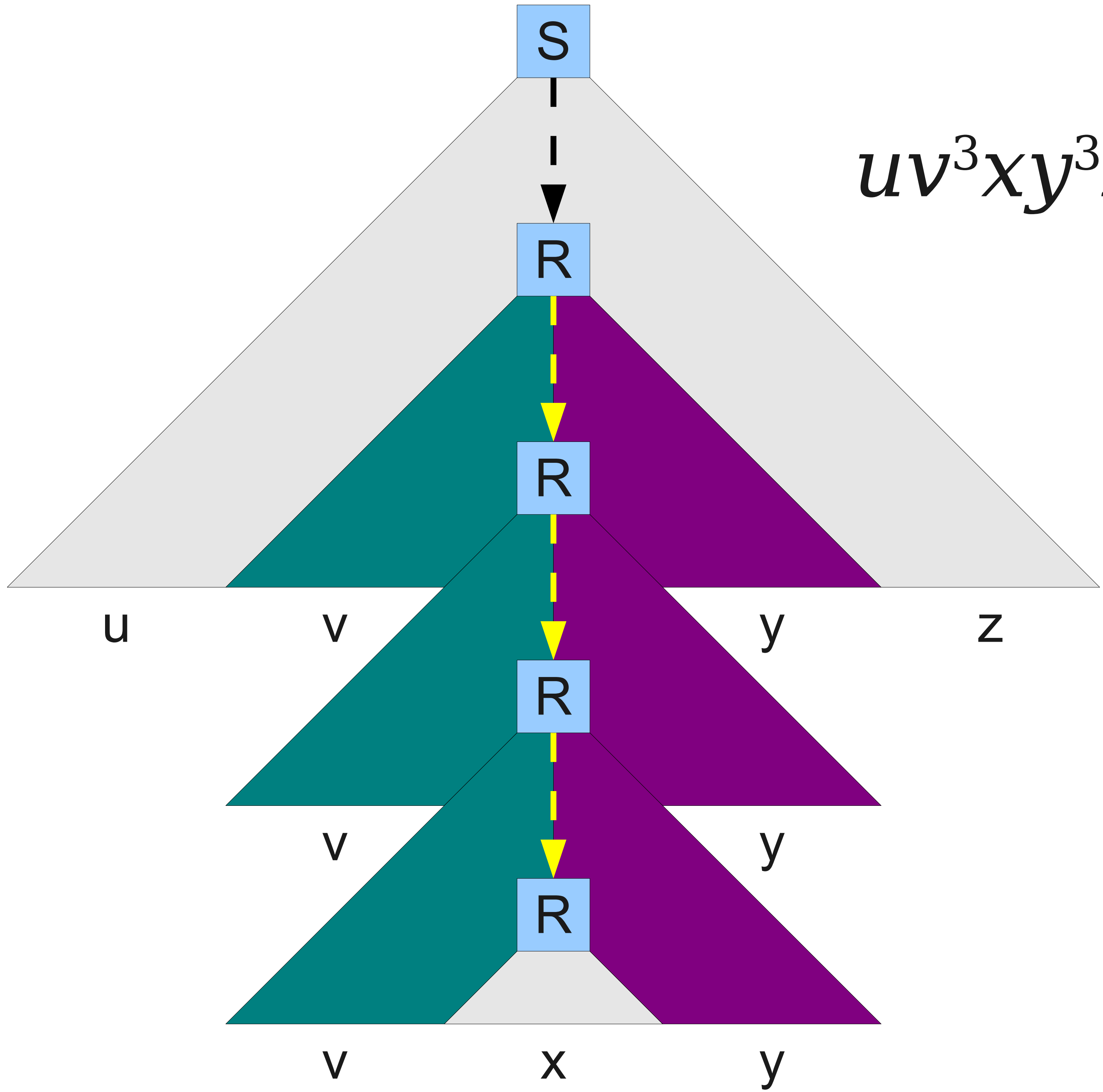




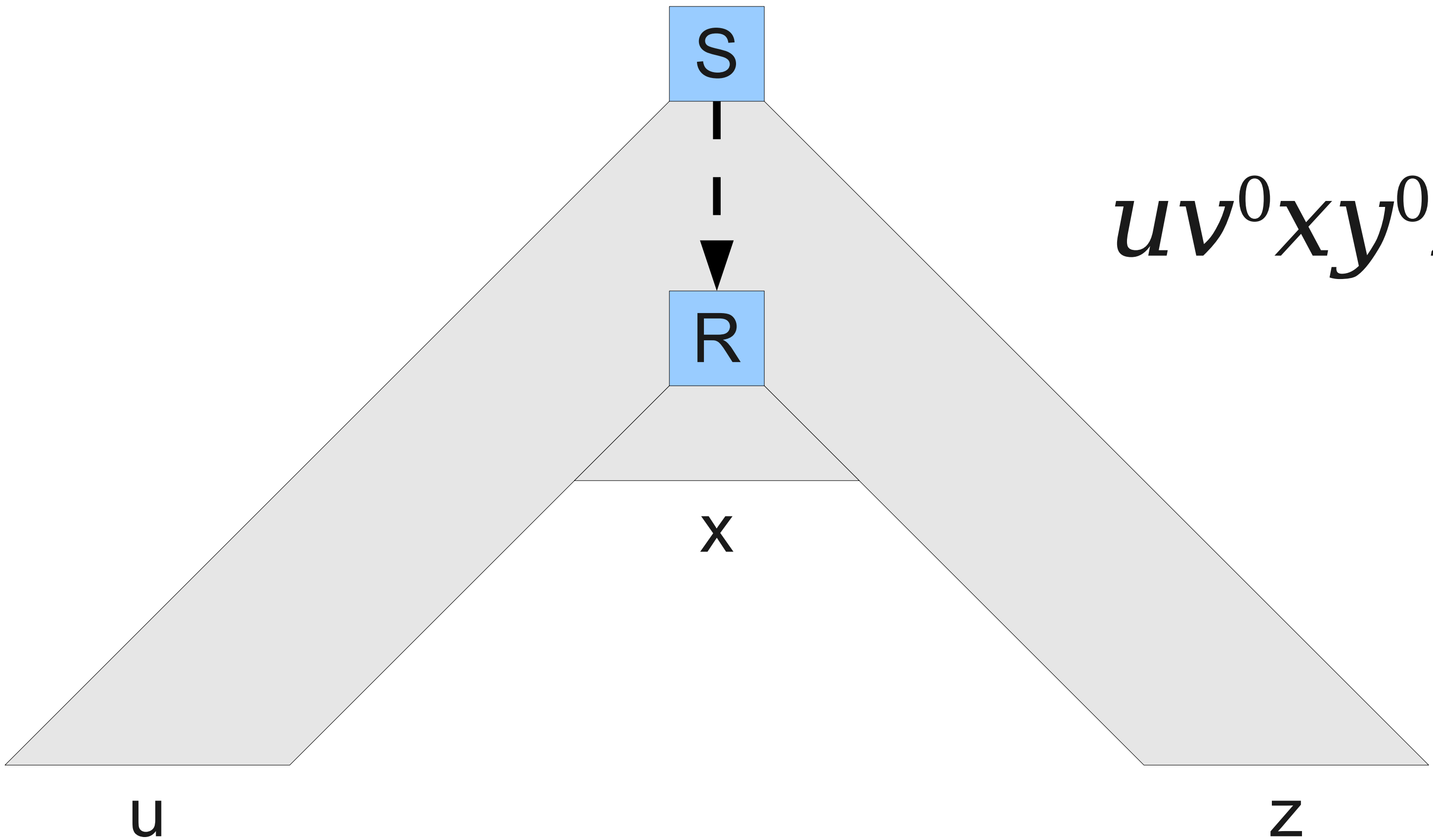
$$uvxyz \in L$$



$$uv^2xy^2z \in L$$



$$uv^3xy^3z \in L$$



$$uv^0xy^0z \in L$$

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$, w can be broken into five pieces,

$|vxy| \leq n$, where the middle three pieces aren't too long,

$|vy| > 0$ where the 2nd and 4th pieces aren't both empty, and

$uv^ixy^iz \in L$ where the 2nd and 4th pieces can be replicated 0 or more times

Note that we pump both v and y at the same time, not just one or the other.

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$, w can be broken into five pieces,

$|vxy| \leq n$, where the middle three pieces aren't too long,

$|vy| > 0$ where the 2nd and 4th pieces aren't both empty, and

$uv^ixy^iz \in L$ where the 2nd and 4th pieces can be replicated 0 or more times

The two strings to pump, collectively, cannot be too long.

They also must be close to one another.

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$, w can be bro

$|vxy| \leq n$, where the middle three pieces aren't too long,

$|vy| > 0$ where the 2nd and 4th pieces aren't both empty, and

$uv^ixy^iz \in L$ where the 2nd and 4th pieces can be replicated 0 or more times

The pumping length is not simple; see Sipser for details.

The Pumping Lemma Game

$L = \{w \in \{0,1,2\}^* \mid w$ has the same number of 0s, 1s, 2s$\}$

ADVERSARY	YOU
Maliciously choose pumping length n.	
	Cleverly choose a string $w \in L$, $\lvert w \rvert \geq n$
Maliciously split $w = uvxyz$, with $\lvert vy \rvert > 0$ and $\lvert vxy \rvert \leq n$	
	Cleverly choose k such that $uv^kxy^kz \notin L$
Grrr! Aaaargh!	

$0^n1^n2^n$

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s. Similarly, if vxy has no 0s in it $uv^2xy^2z \notin L$ because it contains more 1s or 2s than 0s.

In either case, we contradict the pumping lemma. Thus our assumption must have been wrong, so L is not a CFL. ■

Using the Pumping Lemma

- Keep the following in mind when using the context-free pumping lemma when $w = uvxyz$:
  - Both v and y must be pumped at the same time.
  - v and y need not be contiguous in the string.
  - One of v and y may be empty.
  - vxy may be anywhere in the string.
- I **strongly suggest** reading through Sipser to get a better sense for how these proofs work.

Non-CFLs

- Regular languages cannot count once:

  $\{ \mathbf{0}^n\mathbf{1}^n \mid n \in \mathbb{N} \}$ is not regular.

- CFLs cannot count *twice*:
  - $\{ \mathbf{0}^n\mathbf{1}^n\mathbf{2}^n \mid n \in \mathbb{N} \}$ is not context-free.
- A finite number of states cannot count arbitrarily high.
- With a single stack and finite states, cannot track two arbitrary quantities.

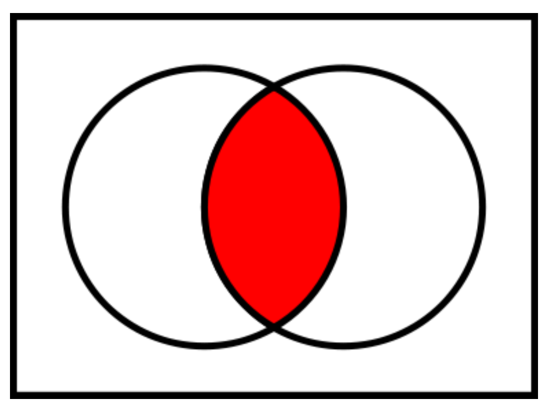
(Non) Closure Properties of CFLs

(Non) Closure Properties of CFLs

- Now that we have a single non-context-free language, we can prove that CFLs are not closed under certain operations.
- Let $L_1 = \{ \mathtt{0}^n\mathtt{1}^n\mathtt{2}^m \mid n, m \in \mathbb{N} \}$
- Let $L_2 = \{\mathtt{0}^m\mathtt{1}^n\mathtt{2}^n \mid n, m \in \mathbb{N} \}$
- Both of these languages are context-free.
  - Can either find an explicit CFG, or note that these languages are the concatenation of two CFLs.
- But $L_1 \cap L_2 = \{ \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \in \mathbb{N} \}$, which is not a CFL.
- **Context-free languages are not closed under intersection**.

(Non) Closure under Complement

- Recall that if L is regular, $\overline{L}$ is regular as well.
- However, if L is context-free, $\overline{L}$ may not be a context-free language.
- Intuition: Using union and complement, we can construct the intersection.



$$\overline{\overline{L_1} \cup \overline{L_2}}$$

(Non) Closure under Subtraction

- ***Theorem:*** If L_1 and L_2 are regular, $L_1 - L_2$ is regular as well.
- However, if L_1 and L_2 are context-free, $L_1 - L_2$ may not be context-free.
- Intuition: We can construct the complement from the difference.
  - Σ^* is context-free because it is regular.
  - But $\Sigma^* - L = \overline{L}$, which may not be context-free.

Summary of CFLs

- CFLs are strictly more powerful than the regular languages.
- CFLs can be described by CFGs (generation) or PDAs (recognition).
- CFGs encompass two classes of languages – deterministic and nondeterministic CFLs.
- Context-free languages have a pumping lemma just as regular languages do.

Problem Session

- Weekly problem session meets tonight at 7PM in 380-380X.
  - Covers CFLs and their limits.
- Optional, but highly recommended!

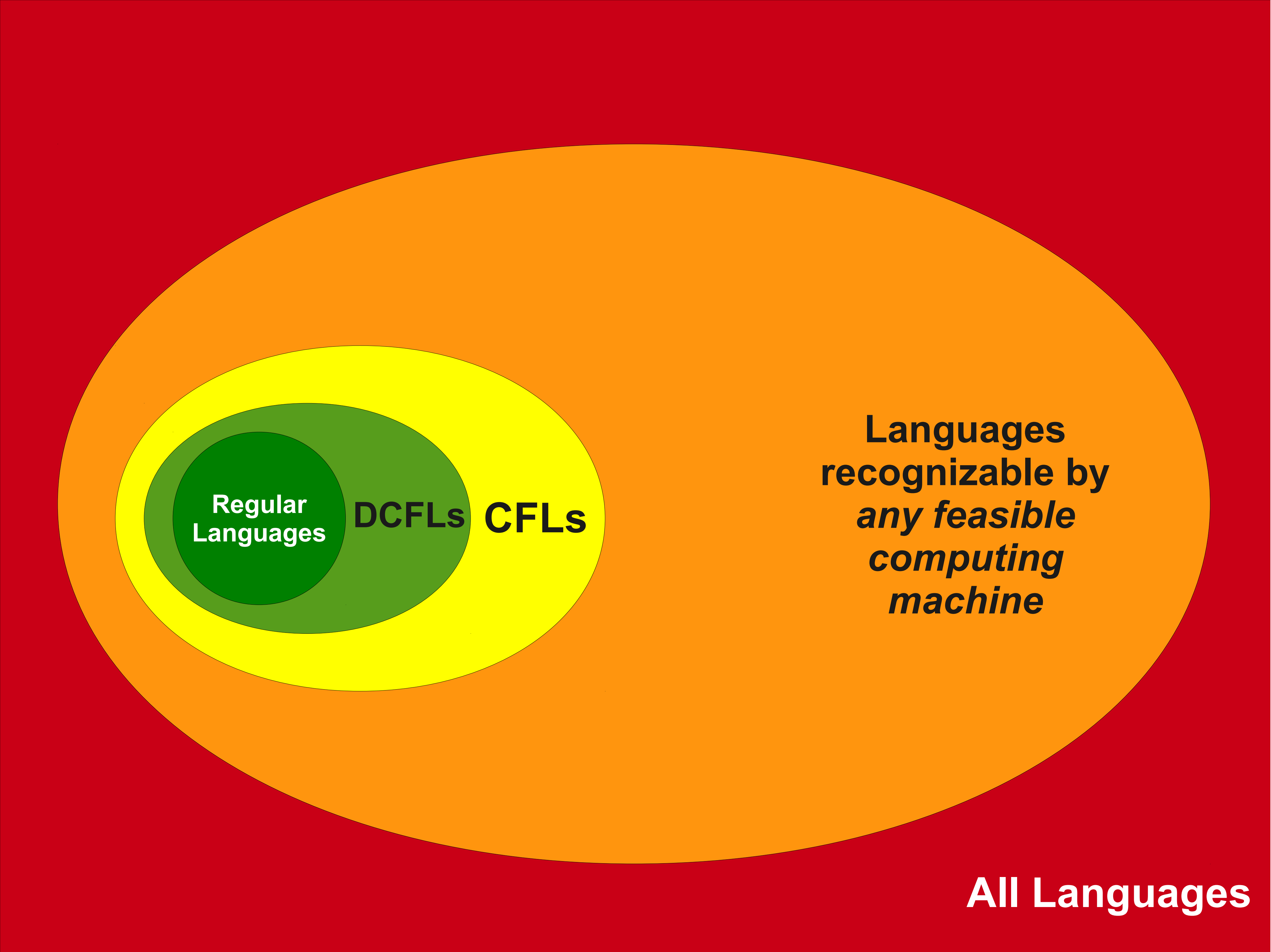
Midterm and Problem Set 4 Graded

Will be distributed at end of lecture.
After that, pick up at my office (Gates 178).

Beyond CFLs

Computability Theory

- **Finite automata** represent computers with bounded memory.
  - They accept precisely the **regular languages**.
- **Pushdown automata** represent computers with a weak infinite memory.
  - They accept precisely the **context-free languages**.
- Regular and context-free languages are comparatively weak.


Regular Languages
DCFLs
CFLs
Languages recognizable by *any feasible computing machine*
All Languages

That same drawing, to scale.

Defining Computability

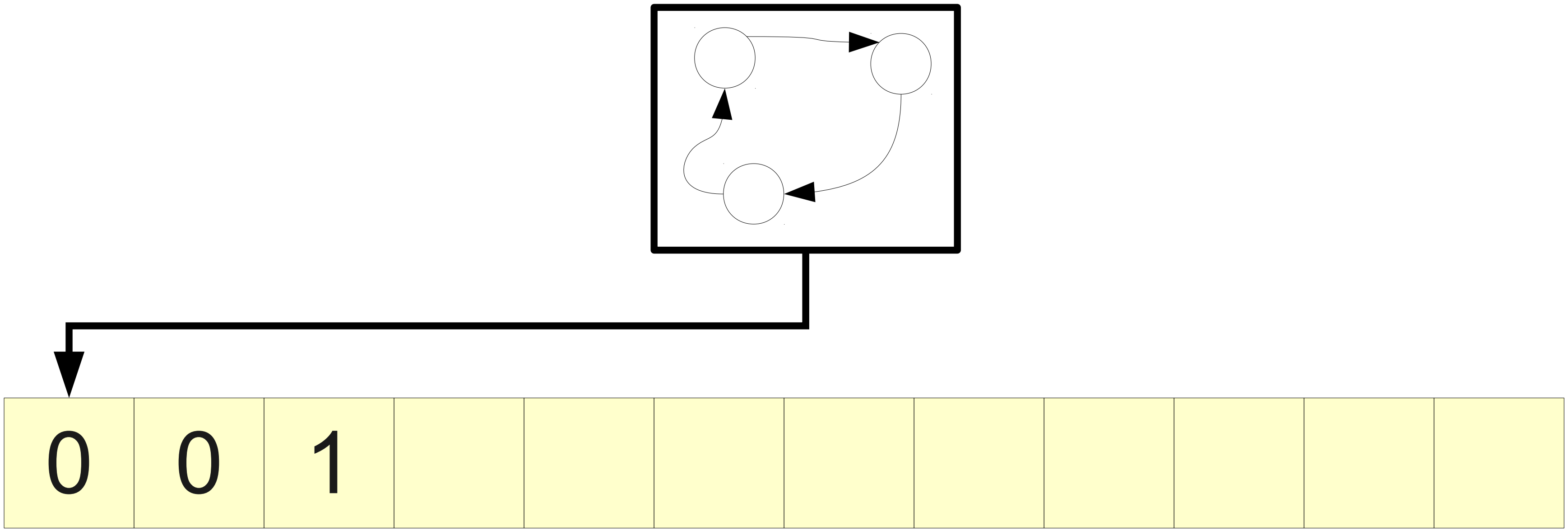
- In order to talk about what languages we could ever hope to recognize with a computer, we need to formalize our model of computation with an automaton.
- The standard automaton for this job is the **Turing machine**, named after Alan Turing, the "Father of Computer Science."

A Better Memory Device

- The pushdown automaton used a (potentially infinite) stack as its memory device.
- This severely limits how the memory can be used:
  - Accessing old data only possible after discarding old data.
  - Can't keep track of multiple unbounded quantities.

A Better Memory Device

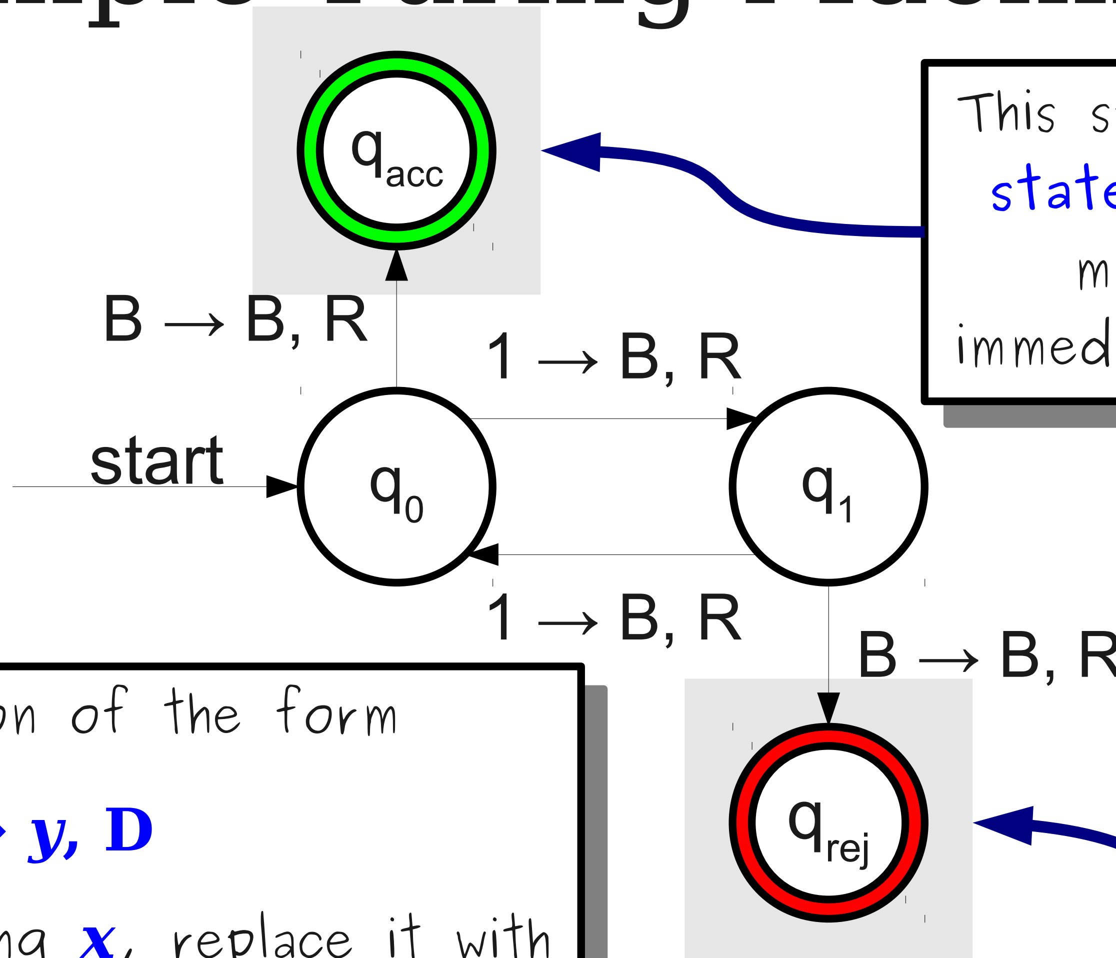
- A **Turing machine** is a finite automaton equipped with an **infinite tape** as its memory.
- The tape begins with the input to the machine written on it, followed by infinitely many blank cells.
- The machine has a **tape head** that can read and write a single memory cell at a time.



The Turing Machine

- A Turing machine consists of three parts:
  - A **finite-state control** used to determine which actions to take,
  - an **infinite tape** serving as both input and scratch space, and
  - a **tape head** that can read and write the tape and move left or right.
- At each step, the Turing machine
  - Replaces the contents of the current cell with a new symbol (which could optionally be the same symbol as before),
  - Changes state, and
  - Moves the tape head to the left or to the right.

A Simple Turing Machine



This special **accept state** causes the machine to immediately accept.

Each transition of the form

x → y, D

means "upon reading **x**, replace it with symbol **y** and move the tape head in direction **D** (which is either **L** or **R**). The letter **B** represents the blank symbol.

This special **reject state** causes the machine to immediately reject.

Acceptance

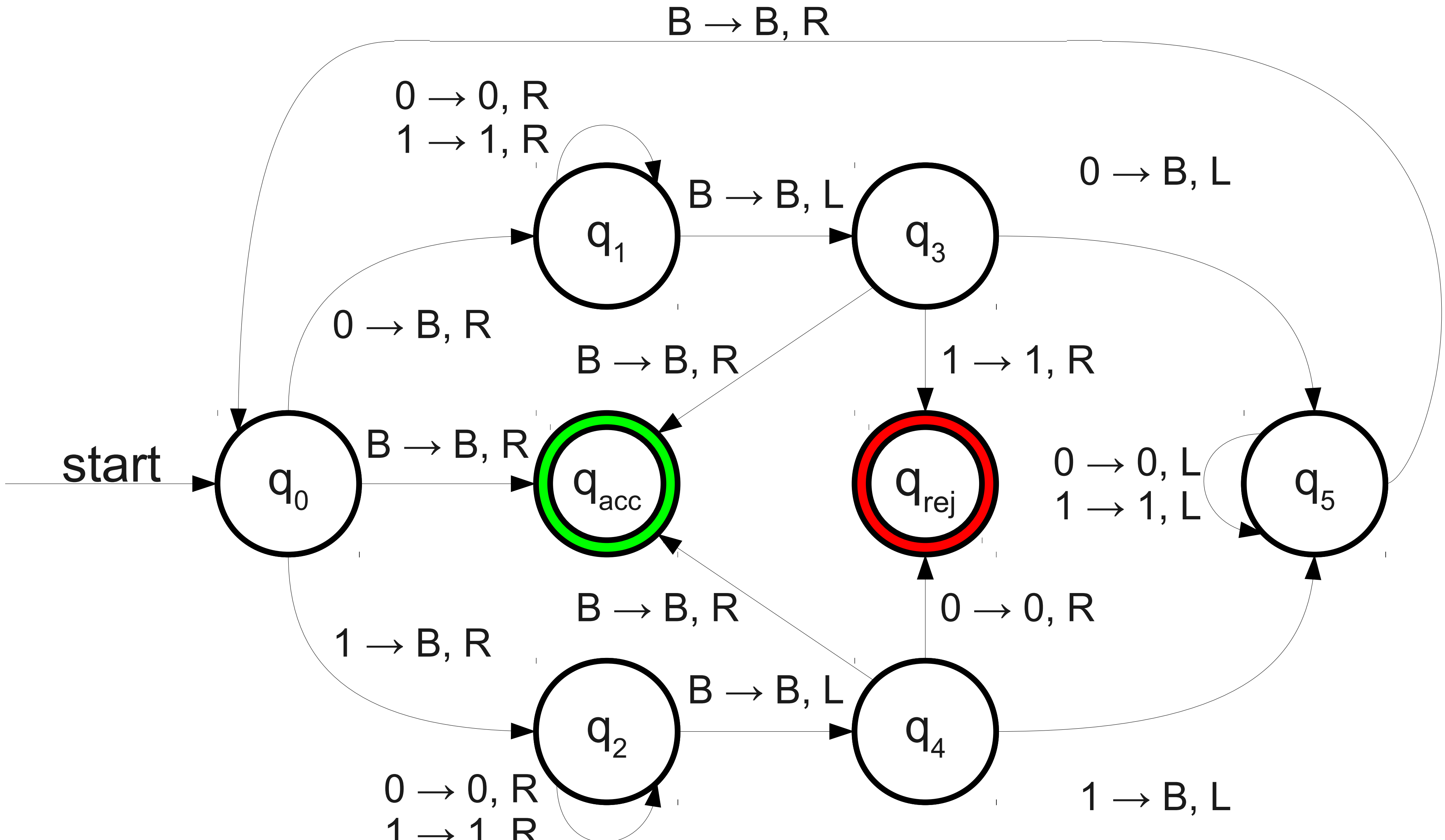
- Unlike the automata that we've seen before, the Turing machine can revisit characters from the input.
- The machine decides when it terminates, rather than stopping when no input is left.
- The Turing machine accepts if it enters a special **accept state**. It rejects if it enters a special **reject state**.
- **Turing machines can loop forever**.
  - More on that later...

A More Powerful Turing Machine

- Let $\Sigma = \{\texttt{0}, \texttt{1}\}$ and let

$$PALINDROME = \{\ w \in \Sigma^* \mid w \text{ is a palindrome}\}$$

- We can build a TM for *PALINDROME* as follows:
  - Look at the leftmost character of the string.
  - Scan across the tape until we find the end of the string.
  - If the last character doesn't match, reject the input.
  - Sweep back to the left of the tape and repeat.
  - If every character becomes matched, accept.



B → B, R
0 → 0, R
1 → 1, R
B → B, L
0 → B, L
q_1
q_3
0 → B, R
B → B, R
1 → 1, R
start
q_0
B → B, R
q_{acc}
q_{rej}
0 → 0, L
1 → 1, L
q_5
B → B, R
0 → 0, R
1 → B, R
B → B, L
q_2
q_4
0 → 0, R
1 → 1, R
1 → B, L

A More Sane Representation

	0			1			B		
q_0	B	R	q_1	B	R	q_2	B	R	q_{acc}
q_1	0	R	q_1	1	R	q_1	B	L	q_3
q_2	0	R	q_2	1	R	q_2	B	L	q_4
q_3	B	L	q_5	1	R	q_{rej}	B	R	q_{acc}
q_4	0	R	q_{rej}	B	L	q_5	B	R	q_{acc}
q_5	0	L	q_5	1	L	q_5	B	R	q_0

Turing Machines, Formally

- A **Turing machine** is an 8-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}, B)$, where
  - Q is a finite set of states,
  - Σ is a finite **input alphabet**,
  - Γ is a finite **tape alphabet**, with $\Sigma \subseteq \Gamma$,
  - $\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L\}$ is the **transition function**,
  - $q_0 \in Q$ is the **start state**,
  - $q_{acc} \in Q$ is the **accept state**,
  - $q_{rej} \in Q$, $q_{rej} \neq q_{acc}$, is the **reject state**, and
  - $B \in \Gamma - \Sigma$ is the **blank symbol**.

The Language of a Turing Machine

- The **language of a TM** M is the set

  $$\mathscr{L}(\mathrm{M}) = \{\ w \in \Sigma^* \mid M \text{ enters } \mathrm{q}_{\mathrm{acc}} \text{ when run on } w\ \}$$

- If there is a TM M such that $\mathscr{L}(M) = L$, we say that L is **Turing-recognizable**.
  - "**Recognizable**" for short.
  - These languages are sometimes called **recursively enumerable**.
- Any regular language is recognizable (why?)
- Harder fact: Any context-free language is recognizable.

Programming Turing Machines

Programming Turing Machines

- Let's begin with a simple language over $\Sigma = \{\texttt{0}, \texttt{1}\}$:
- $BALANCE = \{ w \in \Sigma^* \mid w \text{ contains the same number of } \texttt{0}\text{s and } \texttt{1}\text{s} \}$
- How might we build a TM for *BALANCE*?

The Intuition

- Match the first symbol on the tape with the next available symbol that matches it.
- Match the first symbol on the tape with the next available symbol that matches it.
- Repeat until no symbols are left.
- If everything matches, we're done.
- If there is a mismatch, report failure.

TM for *BALANCE*

<table>
<tr><th></th><th colspan="3">0</th><th colspan="3">1</th><th colspan="3">B</th><th colspan="3">x</th></tr>
<tr><th>q_{st}</th><td>B</td><td>R</td><td>q_{m0}</td><td>B</td><td>R</td><td>q_{m1}</td><td colspan="3">Accept</td><td>x</td><td>R</td><td>q_{st}</td></tr>
<tr><th>q_{m0}</th><td>0</td><td>R</td><td>q_{m0}</td><td>x</td><td>L</td><td>q_{ret}</td><td colspan="3">Reject</td><td>x</td><td>R</td><td>q_{m0}</td></tr>
<tr><th>q_{m1}</th><td>x</td><td>L</td><td>q_{ret}</td><td>1</td><td>R</td><td>q_{m1}</td><td colspan="3">Reject</td><td>x</td><td>R</td><td>q_{m1}</td></tr>
<tr><th>q_{ret}</th><td>0</td><td>L</td><td>q_{ret}</td><td>1</td><td>L</td><td>q_{ret}</td><td>B</td><td>R</td><td>q_{st}</td><td>x</td><td>L</td><td>q_{ret}</td></tr>
</table>

The Key Insight

- Our construction worked because we could make the finite-state control hold extra information (which symbol we had matched).
- ***General TM design trick***: Treat the finite state control as a combination control/finite memory.
- Can hold any finite amount of information by just replicating important states the appropriate number of times.

A More Elaborate Language

- Consider $\Sigma = \{ \texttt{1}, \texttt{×}, \texttt{=} \}$ and the language

$$\text{MULTIPLY} = \{ \texttt{1}^n \texttt{×} \texttt{1}^m \texttt{=} \texttt{1}^{mn} \mid m, n \in \mathbb{N} \}$$

- This language is neither regular nor context-free, but it is recursively enumerable.
- How would we build a TM for it?

A Turing Machine Subroutine

- A **subroutine** in a TM is state that, when entered:
  - Performs some specific task on the tape, then
  - Terminates in a well-specified state.
- Complex Turing machines can be broken down into smaller subroutines as follows:
  - The start state fires off the first subroutine.
  - After the first subroutine terminates, the next begins.
  - (etc.)
  - The machine may accept or reject at any point.

Key Idea: Subroutines

- Checking whether a string is in *MULTIPLY* requires several different steps:
    - Check that the string is formatted correctly.
    - Compute $m \times n$.
    - Confirm that $m \times n$ matches what's given.
- Let's design a subroutine for each of these.

Validating the Input

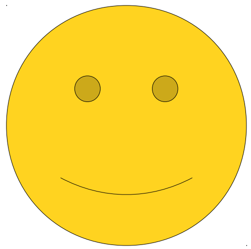
- High-level idea:
  - Shift the input over by one step.

1	×	1	=	1	

	1	×	1	=	1

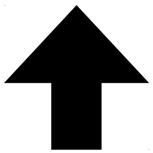
  - Check the structure of the input.

	1	×	1	=	1



  - End up in a new state looking at the first character of the input if successful.

	1	×	1	=	1



Step One: Shift the Input

	1			×			=			B		
q_s	B	R	q_1	B	R	$q_\times$	B	R	$q_=$	B	R	**D**
q_1	1	R	q_1	1	R	$q_\times$	1	R	$q_=$	1	L	q_R
$q_\times$	×	R	q_1	×	R	$q_\times$	×	R	$q_=$	×	L	q_R
$q_=$	=	R	q_1	=	R	$q_\times$	=	R	$q_=$	=	L	q_R
q_R	1	L	q_R	×	L	q_R	=	L	q_R	B	R	**D**

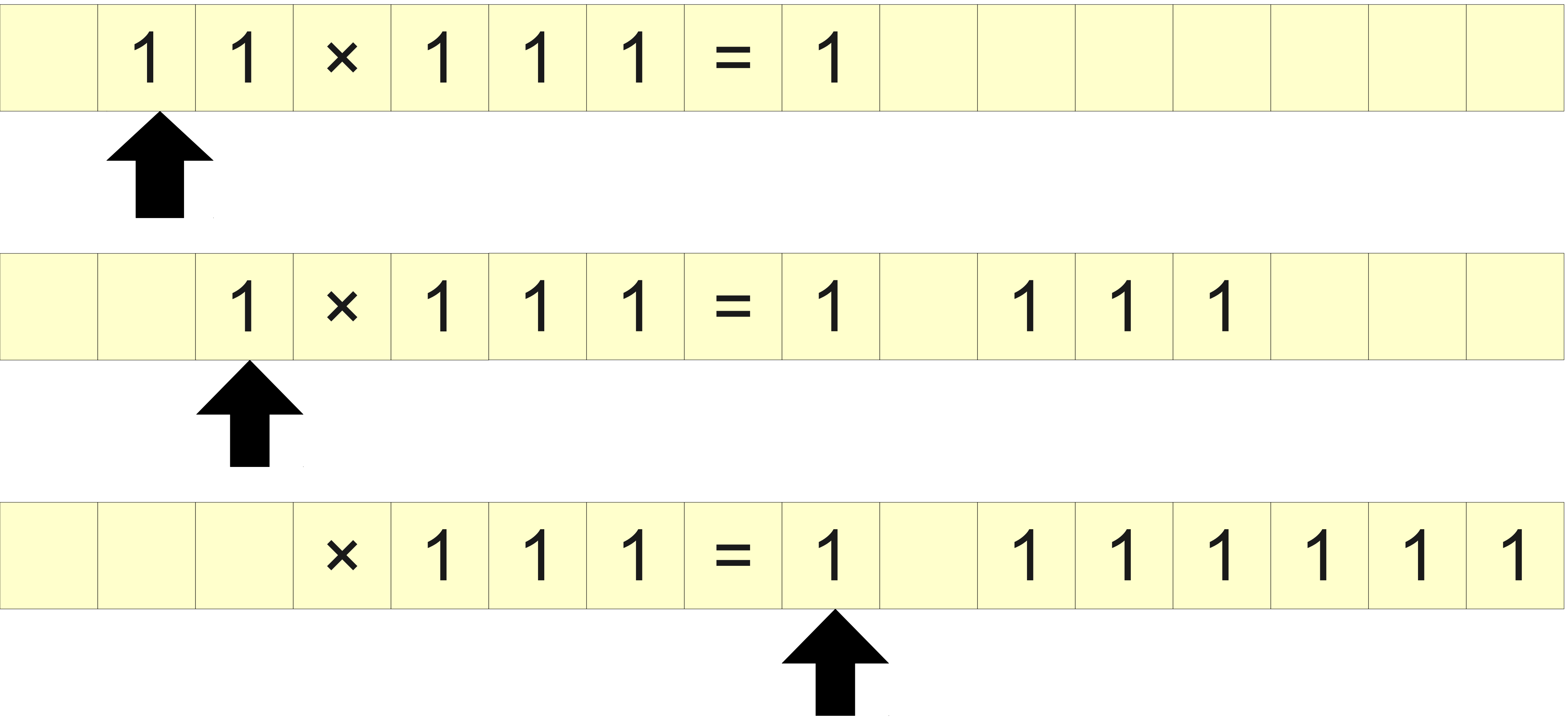
Step Two: Verify the Input

	1			×			=			B		
q_{st}	1	R	q_{st}	×	R	$q_{×}$	Reject			Reject		
$q_{×}$	1	R	$q_{×}$	Reject			=	R	$q_{=}$	Reject		
$q_{=}$	1	R	$q_{=}$	Reject			Reject			B	L	q_L
q_L	1	L	q_L	×	L	q_L	=	L	q_L	B	R	**D**

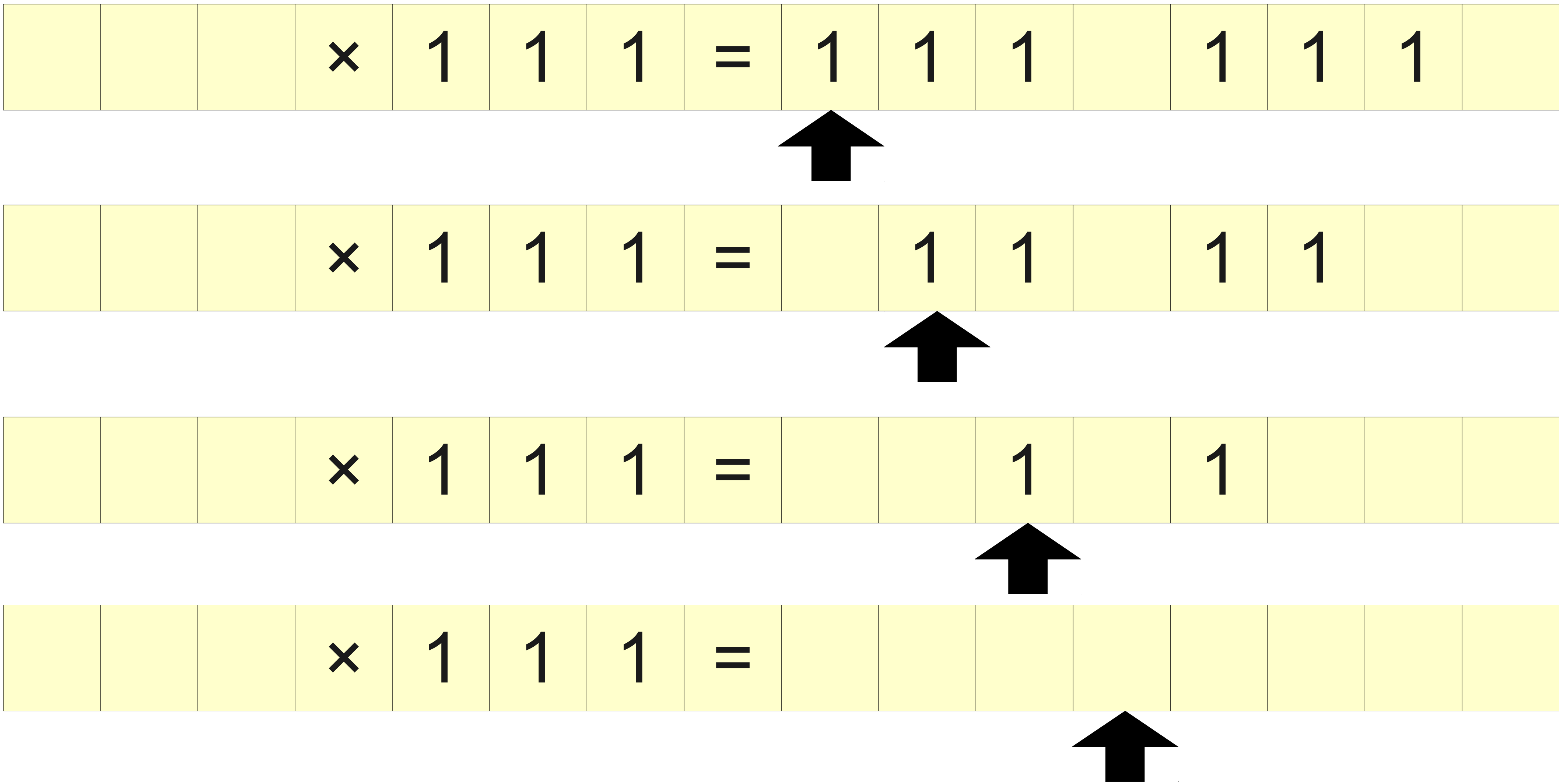
Putting it Together: Shift/Verify

	1			×			=			B		
q_s	B	R	q_1	B	R	$q_\times$	B	R	$q_=$	B	R	q_{s2}
q_1	1	R	q_1	1	R	$q_\times$	1	R	$q_=$	1	L	q_R
$q_\times$	×	R	q_1	×	R	$q_\times$	×	R	$q_=$	×	L	q_R
$q_=$	=	R	q_1	=	R	$q_\times$	=	R	$q_=$	=	L	q_R
q_R	1	L	q_R	×	L	q_R	=	L	q_R	B	R	q_{s2}
q_{s2}	1	R	q_{s2}	×	R	$q_{\times 2}$	Reject			Reject		
$q_{\times 2}$	1	R	$q_{\times 2}$	Reject			=	R	$q_{=2}$	Reject		
$q_{=2}$	1	R	$q_{=2}$	Reject			Reject			B	L	q_{L2}
q_{L2}	1	L	q_{L2}	×	L	q_{L2}	=	L	q_{L2}	B	R	**D**

Step Three: Doing the Multiply



Step Four: Checking the Multiply



Why This Matters

- TMs can solve a large class of problems, but they can be enormously complicated.
- We now have two tricks for designing TMs:
  - Constant storage
  - Subroutines
- We can use these tricks to show that if we can get each individual piece working, we can solve a large problem with a TM.

Next Time

- **Programming Turing Machines**
  - A cleaner way to think about TMs.
- **The Power of Turing Machines**
  - Just how much expressive power do TMs have?