Pushdown automata and context-free languages

Sides based on material by Kani, Erickson, Chekuri, et. al.

All mistakes are my own! - Ivan Abraham (Fall 2024)

Image by ChatGPT (probably collaborated with DALL-E)

Introduction

Will this code execute successfully?

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    int a = 2;
    int b = 5 ;
}
```

Introduction

Will this code execute successfully? **YES**

main.c

Share Run

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    int a = 2;
    int b = 5 ;
}
```

Output Clear

```
/tmp/3w7RzFLskv.o


=== Code Execution Successful ===
```

Introduction

Will this code execute successfully?

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    int a = 2;
    int b =  ;
}
```

Introduction

Will this code execute successfully? **NO**

main.c   Share   Run

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    int a = 2;
    int b =  ;
}
```

Output   Clear

```
/tmp/Zqon05DwrB.c: In function 'main':
ERROR!
/tmp/Zqon05DwrB.c:6:14: error: expected expression
    before ';' token
    6 |     int b =  ;
      |              ^


=== Code Exited With Errors ===
```

Introduction

Will this code execute successfully? **NO**

main.c Share Run

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    int a = 2;
    int b =  ;
}
```

Output Clear

```
/tmp/Zqon05DwrB.c: In function 'main':
ERROR!
/tmp/Zqon05DwrB.c:6:14: error: expected expression
    before ';' token
    6 |     int b =  ;
      |              ^


=== Code Exited With Errors ===
```

What was the compiler expecting?

What can an arithmetic expression be?

- **int - ** A single number.

What can an arithmetic expression be?

- **int** - A single number.

$$\frac{\textbf{int}}{\textbf{Expr}}$$

What can an arithmetic expression be?

- **int** - A single number.

$$\frac{\textbf{int}}{\textbf{Expr}}$$

- **Expr Op Expr** - Two expressions joined by an operator.

What can an arithmetic expression be?

- **int** - A single number.

int

Expr

- **Expr Op Expr** - Two expressions joined by an operator.

int	+	
Expr	Op	Expr

What can an arithmetic expression be?

- **int** - A single number.

int
———
Expr

- **Expr Op Expr** - Two expressions joined by an operator.

int	+	
Expr	Op	Expr

- **Recursive Expression**

What can an arithmetic expression be?

- **int** - A single number.

int
Expr

- **Expr Op Expr** - Two expressions joined by an operator.

int | + |
Expr | Op | Expr

- **Recursive Expression**

int | + |
Expr | Op | Expr | Op | Expr

Arithmetic expressions

- Here is one way to express these rules

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**
- **Expr** → **Expr Op Expr**

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**
- **Expr** → **Expr Op Expr**

This is called a ***production rule***. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**
- **Expr** → **Expr Op Expr**
- **Expr** → **(Expr)**

This is called a ***production rule***. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**
- **Expr** → **Expr Op Expr**
- **Expr** → **(Expr)**
- **Op** → **+** | **-** | **×** | **/**

This is called a ***production rule***. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

Arithmetic expressions

- Here is one way to express these rules
- **Expr** → **int**
- **Expr** → **Expr Op Expr**
- **Expr** → **(Expr)**
- **Op** → **+** | **-** | **×** | **/**

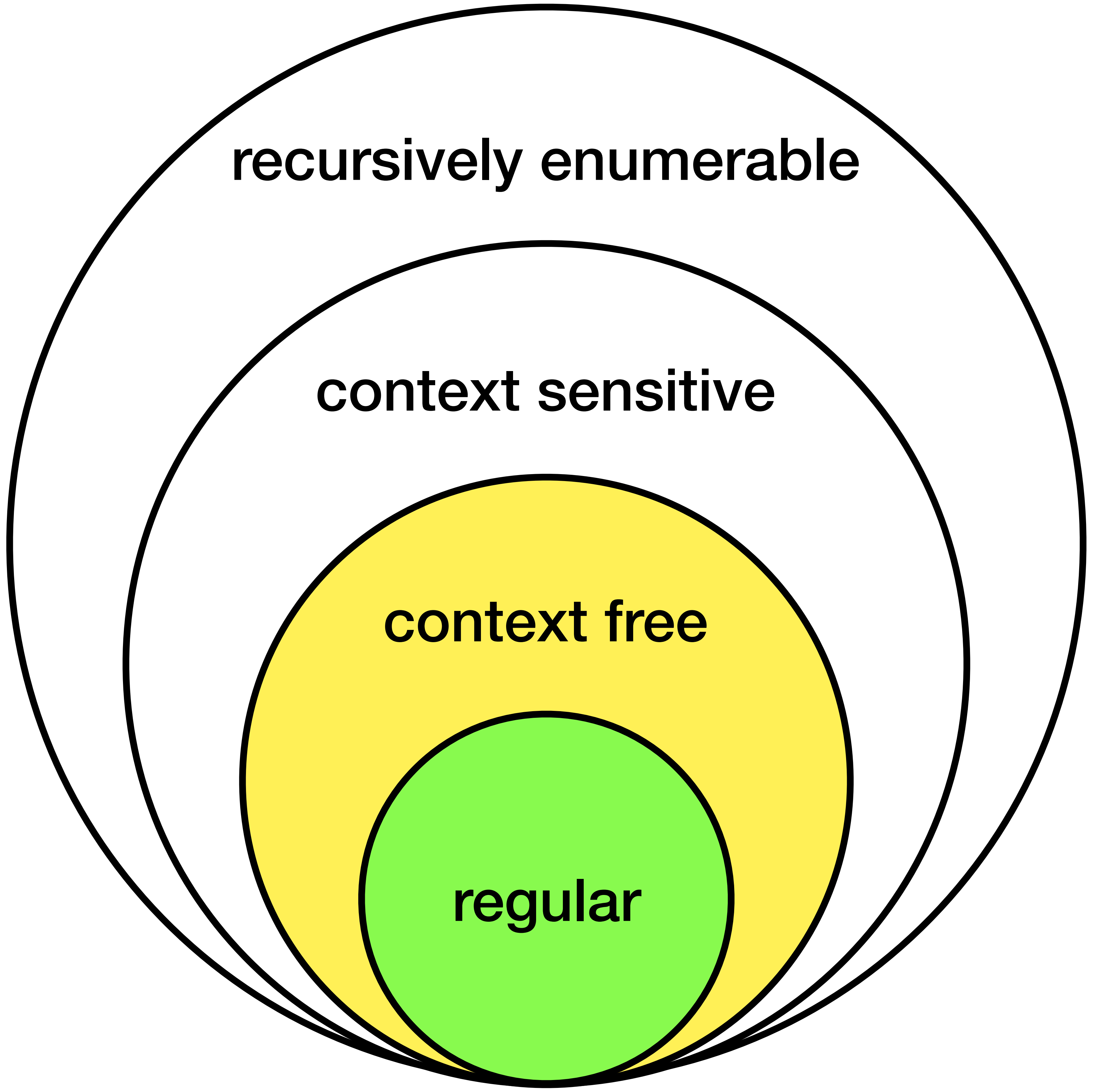
This is called a ***production rule***. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

This one says "if you see **Op**, you can replace it with **+** or **-** or **×** or **/**

Grammar - rules for a language

A ***context-free grammar*** (or ***CFG***) is a recursive set of rules that define a language.

Definition:

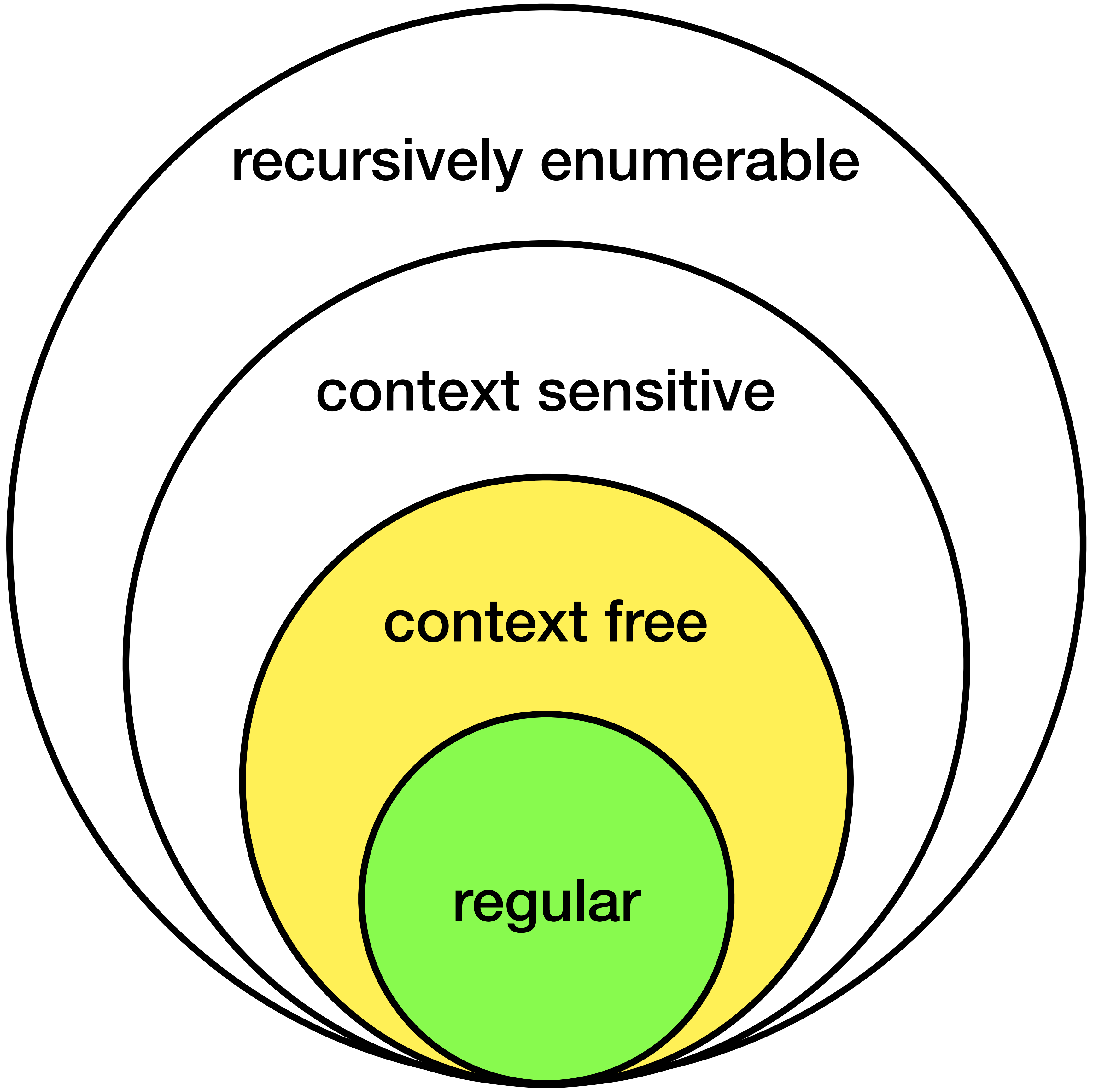


Grammar - rules for a language

A ***context-free grammar*** (or ***CFG***) is a recursive set of rules that define a language.

Definition:

A **CFG** is a quadruple $G = (V, T, P, S)$ where



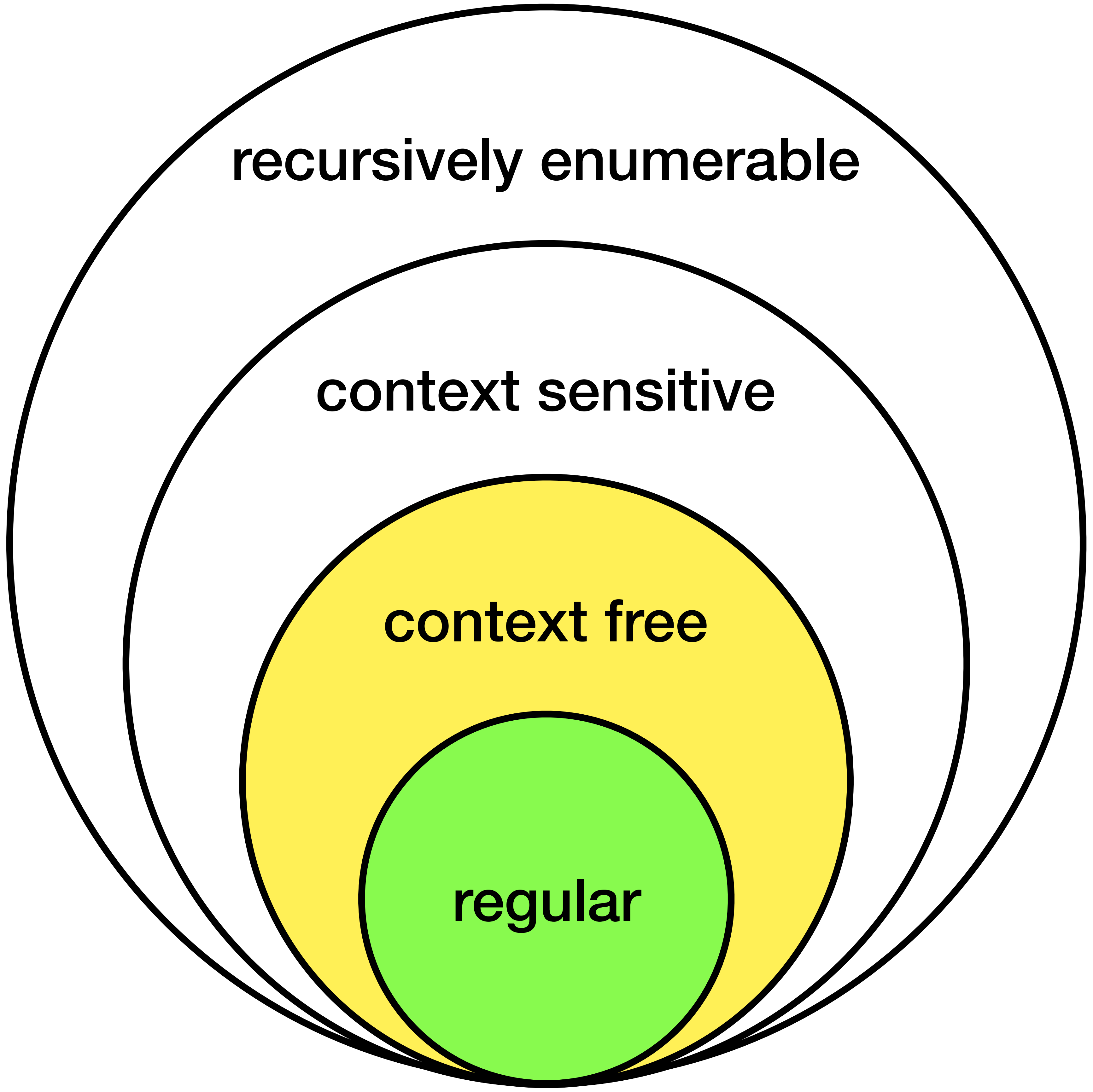
Grammar - rules for a language

A ***context-free grammar*** (or ***CFG***) is a recursive set of rules that define a language.

Definition:

A **CFG** is a quadruple $G = (V, T, P, S)$ where

G = (*Variables*, *Terminals*, *Productions*, *Start Var*)



Context Free Grammar

Definition: A **CFG** is a quadruple $G = (V, T, P, S)$

Context Free Grammar

Definition: A **CFG** is a quadruple $G = (V, T, P, S)$

- V is a finite set of ***non-terminal*** (***variable***) ***symbols***

Context Free Grammar

Definition: A **CFG** is a quadruple $G = (V, T, P, S)$

- V is a finite set of ***non-terminal*** (***variable***) ***symbols***
- T is a finite set of ***terminal symbols*** (***alphabet***)

Context Free Grammar

Definition: A **CFG** is a quadruple $G = (V, T, P, S)$

- V is a finite set of ***non-terminal*** (***variable***) ***symbols***
- T is a finite set of ***terminal symbols*** (***alphabet***)
- P is a finite set of ***productions***, each of the form $A \to \alpha$ where $A \in V$ and α is a string in $(V \cup T)^*$.
  Formally, $P \subset V \times (V \cup T)^*$

Context Free Grammar

Definition: A **CFG** is a quadruple $G = (V, T, P, S)$

- V is a finite set of ***non-terminal*** (***variable***) ***symbols***
- T is a finite set of ***terminal symbols*** (***alphabet***)
- P is a finite set of ***productions***, each of the form $A \rightarrow \alpha$ where $A \in V$ and α is a string in $(V \cup T)^*$.
  Formally, $P \subset V \times (V \cup T)^*$
- $S \in V$ is a ***start symbol*** .

Context Free Grammar

Example

- $V = \{S\}$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ (*abbrev. for* $S \to \epsilon,\ S \to 0S0,\ S \to 1S1$)

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ (*abbrev. for* $S \to \epsilon,\ S \to 0S0,\ S \to 1S1$)
- $S = S$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ *(abbrev. for* $S \to \epsilon,\ S \to 0S0,\ S \to 1S1$*)*
- $S = S$

S

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \rightarrow \varepsilon \mid 0S0 \mid 1S1\}$ (*abbrev. for* $S \rightarrow \epsilon,\ S \rightarrow 0S0,\ S \rightarrow 1S1$)
- $S = S$

$S \rightarrow 0S0$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ *(abbrev. for* $S \to \epsilon,\ \ S \to 0S0,\ S \to 1S1$ *)*
- $S = S$

$$S \rightarrow 0S0 \rightarrow 01S10$$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \rightarrow \varepsilon \mid 0S0 \mid 1S1\}$ *(abbrev. for* $S \rightarrow \epsilon,\ S \rightarrow 0S0,\ S \rightarrow 1S1$ *)*
- $S = S$

$$S \rightarrow 0S0 \rightarrow 01S10 \rightarrow 011S110$$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ *(abbrev. for* $S \to \epsilon,\ S \to 0S0,\ S \to 1S1$*)*
- $S = S$

$$S \to 0S0 \to 01S10 \to 011S110 \to 011\epsilon110$$

Context Free Grammar

Example

- $V = \{S\}$
- $T = \{0,1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$ (*abbrev. for* $S \to \epsilon,\ S \to 0S0,\ S \to 1S1$)
- $S = S$

cannot produce an odd length string

010

$S \to 0S0$

$\not\downarrow$

?

$$S \to 0S0 \to 01S10 \to 011S110 \to 011\epsilon110 \to 011110$$

Derives relation

Formalism for how strings are derived/generated

Definition: Let $G = (V, T, P, S)$ be a CFG. For strings, $\alpha_1, \alpha_2 \in (V \cup T)*$ we say α_2 *derives from* α_1, denoted by $\alpha_1 \rightsquigarrow \alpha_2$, if there exist strings β, γ, δ in $(V \cup T)*$ such that

Derives relation

Formalism for how strings are derived/generated

Definition: Let $G = (V, T, P, S)$ be a CFG. For strings, $\alpha_1, \alpha_2 \in (V \cup T)^*$ we say α_2 *derives from* α_1, denoted by $\alpha_1 \rightsquigarrow \alpha_2$, if there exist strings β, γ, δ in $(V \cup T)^*$ such that

- $\alpha_1 = \beta A \delta$

Derives relation

Formalism for how strings are derived/generated

Definition: Let $G = (V, T, P, S)$ be a CFG. For strings, $\alpha_1, \alpha_2 \in (V \cup T)^*$ we say α_2 *derives from* α_1, denoted by $\alpha_1 \rightsquigarrow \alpha_2$, if there exist strings β, γ, δ in $(V \cup T)^*$ such that

- $\alpha_1 = \beta A \delta$
- $\alpha_2 = \beta \gamma \delta$

$P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$

Derives relation

Formalism for how strings are derived/generated

Definition: Let $G = (V, T, P, S)$ be a CFG. For strings, $\alpha_1, \alpha_2 \in (V \cup T)^*$ we say α_2 *derives from* α_1, denoted by $\alpha_1 \rightsquigarrow \alpha_2$, if there exist strings β, γ, δ in $(V \cup T)^*$ such that

- $\alpha_1 = \beta A \delta$
- $\alpha_2 = \beta \gamma \delta$
- $A \to \gamma \in P$

Derives relation

Formalism for how strings are derived/generated

Definition: For integers $k \geq 0$, define $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ inductively as follows:

Derives relation

Formalism for how strings are derived/generated

Definition: For integers $k \geq 0$, define $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ inductively as follows:

- $\alpha_1 \overset{0}{\rightsquigarrow} \alpha_2$ if $\alpha_1 = \alpha_2$ Base case

Derives relation

Formalism for how strings are derived/generated

Definition: For integers $k \geq 0$, define $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ inductively as follows:

- $\alpha_1 \overset{0}{\rightsquigarrow} \alpha_2$ if $\alpha_1 = \alpha_2$ Base case
- $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ if $\alpha_1 \rightsquigarrow \beta_1$ and $\beta_1 \overset{k-1}{\rightsquigarrow} \alpha_2$

single step first

Derives relation

Formalism for how strings are derived/generated

Definition: For integers $k \geq 0$, define $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ inductively as follows:

- $\alpha_1 \overset{0}{\rightsquigarrow} \alpha_2$ if $\alpha_1 = \alpha_2$
- $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ if $\alpha_1 \rightsquigarrow \beta_1$ and $\beta_1 \overset{k-1}{\rightsquigarrow} \alpha_2$
- Alternatively, $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ if $\alpha_1 \overset{k-1}{\rightsquigarrow} \beta_1$ and $\beta_1 \rightsquigarrow \alpha_2$

single step
last.

Derives relation

Formalism for how strings are derived/generated

Definition: For integers $k \geq 0$, define $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ inductively as follows:

- $\alpha_1 \overset{0}{\rightsquigarrow} \alpha_2$ if $\alpha_1 = \alpha_2$
- $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ if $\alpha_1 \rightsquigarrow \beta_1$ and $\beta_1 \overset{k-1}{\rightsquigarrow} \alpha_2$
- Alternatively, $\alpha_1 \overset{k}{\rightsquigarrow} \alpha_2$ if $\alpha_1 \overset{k-1}{\rightsquigarrow} \beta_1$ and $\beta_1 \rightsquigarrow \alpha_2$

Finally, we use the notation $\alpha_1 \overset{*}{\rightsquigarrow} \alpha_2$ to mean that α_2 can be derived from α_1. In other words,

$$\alpha_1 \overset{*}{\rightsquigarrow} \alpha_2 \text{ if } \alpha_1 \overset{k}{\rightsquigarrow} \alpha_2 \text{ for some } k$$

"α_1 'becomes' α_2 in zero or more steps"

Context Free Languages

Definition: Let $G = (V, T, P, S)$ be a CFG. Then the *language generated* by G, denoted by $L(G)$ is the set

Context Free Languages

Definition: Let $G = (V, T, P, S)$ be a CFG. Then the *language generated* by G, denoted by $L(G)$ is the set

$$L(G) := \left\{ w \in T^* \ \mid \ S \overset{*}{\rightsquigarrow} w \right\}.$$

Thus, a language L is context-free (called a context-free language or CFL) if it is generated by a context-free grammar.

Context Free Languages

Definition: Let $G = (V, T, P, S)$ be a CFG. Then the *language generated* by G, denoted by $L(G)$ is the set

$$L(G) := \left\{ w \in T^* \mid S \overset{*}{\rightsquigarrow} w \right\}.$$

Thus, a language L is context-free (called a context-free language or CFL) if it is generated by a context-free grammar.

Alternatively, a language L is said to be a CFL, if there exists a CFG G such that $L = L(G)$.

Context Free Languages

Production rule examples

- $L = \{0^n 1^n \mid n \geq 0\}$

- n can be zero

  $S \rightarrow \varepsilon$

- if there is a zero, then there must be a 1

  $S \rightarrow 01$

- Need to repeat "01" as many times as necessary in the where $0 \square 1$

  $S \rightarrow 0S1$

$$S \rightarrow \varepsilon \mid 0S1$$

Context Free Languages

Production rule examples

- $L = \{0^n 1^n \mid n \geq 0\}$
- $L = \{0^n 1^m \mid m \geq n\}$

- n can be zero (and equal to m)

$S \rightarrow \varepsilon$

- n can be same as m (and not zero)

$S \rightarrow 0S1$

- m can be greater than n.

$S \rightarrow S1$

$S \rightarrow \varepsilon \mid 0S1 \mid S1$

what if $m > n$?

$S \rightarrow 1 \mid 0S1 \mid S1$

$S \rightarrow \varepsilon \mid 0S1 \mid SR$

$R \rightarrow 1$

Context Free Languages

Production rule examples

- $L = \{0^n1^n \mid n \geq 0\}$
- $L = \{0^n1^m \mid m \geq n\}$
- $L = \{0^n1^m \mid m, n \geq 0\}$

m, n can be 0

$S \to \varepsilon$.

$S \to 0S$ zero always at beginning

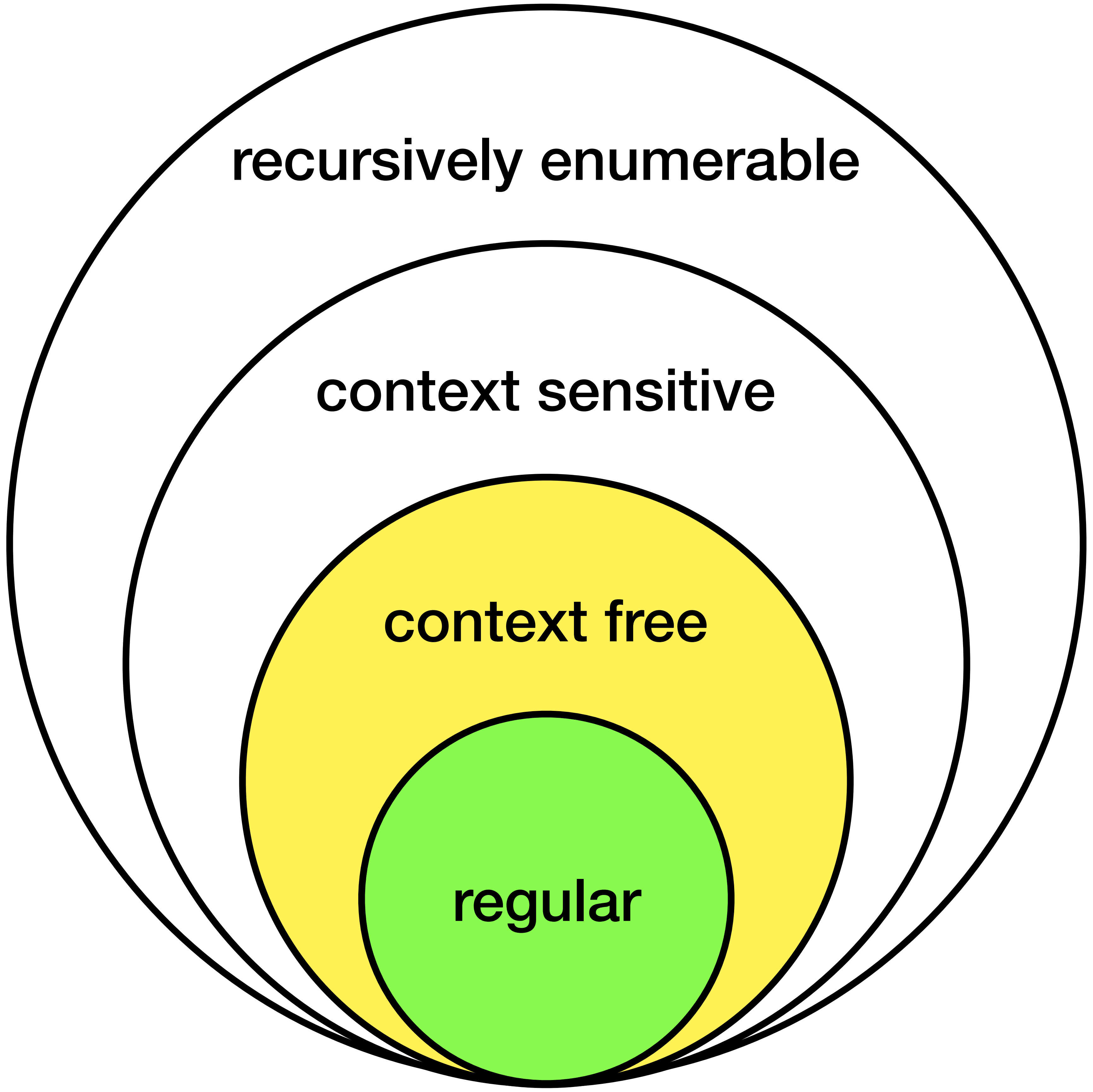
$S \to S1$ one always at end.

$S \to \varepsilon \mid 0S \mid S1$

CFL/CFGs and regular languages

Recall Chomsky Heirarchy

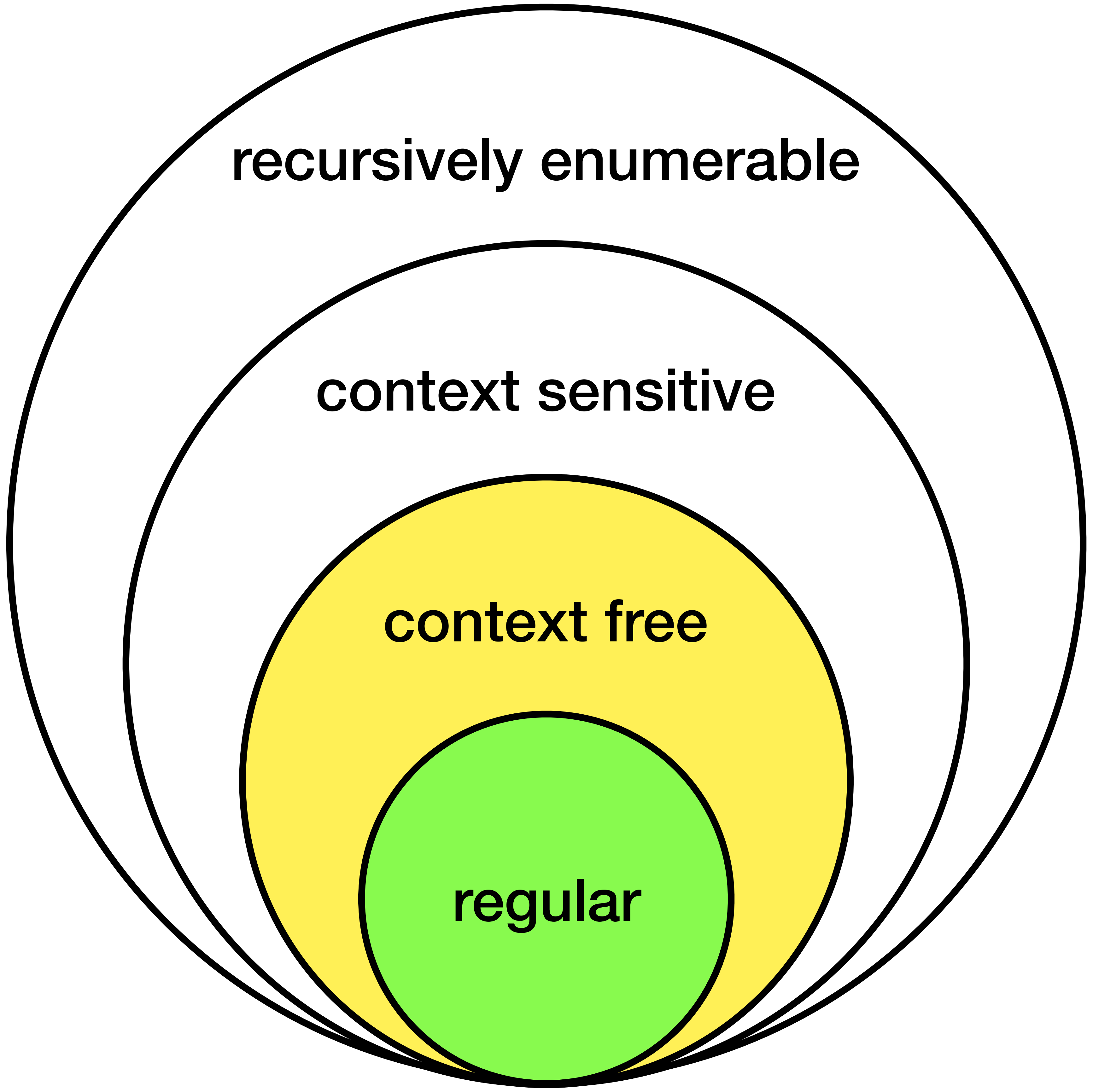
- The picture depicts regular languages as a proper subset of context-free languages.



CFL/CFGs and regular languages

Recall Chomsky Heirarchy

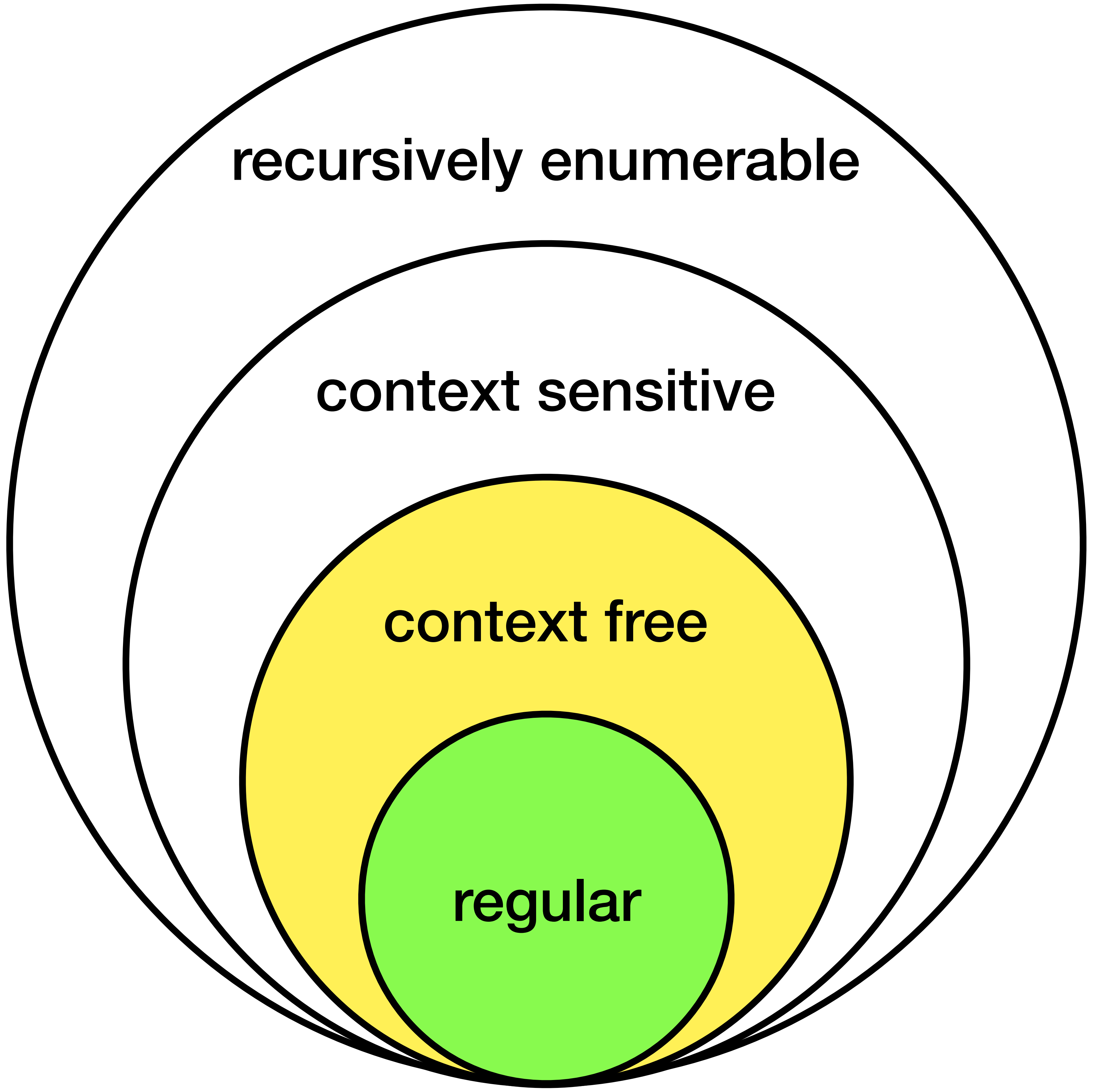
- The picture depicts regular languages as a proper subset of context-free languages.
- Thus, all regular languages are also CFLs.



CFL/CFGs and regular languages

Recall Chomsky Heirarchy

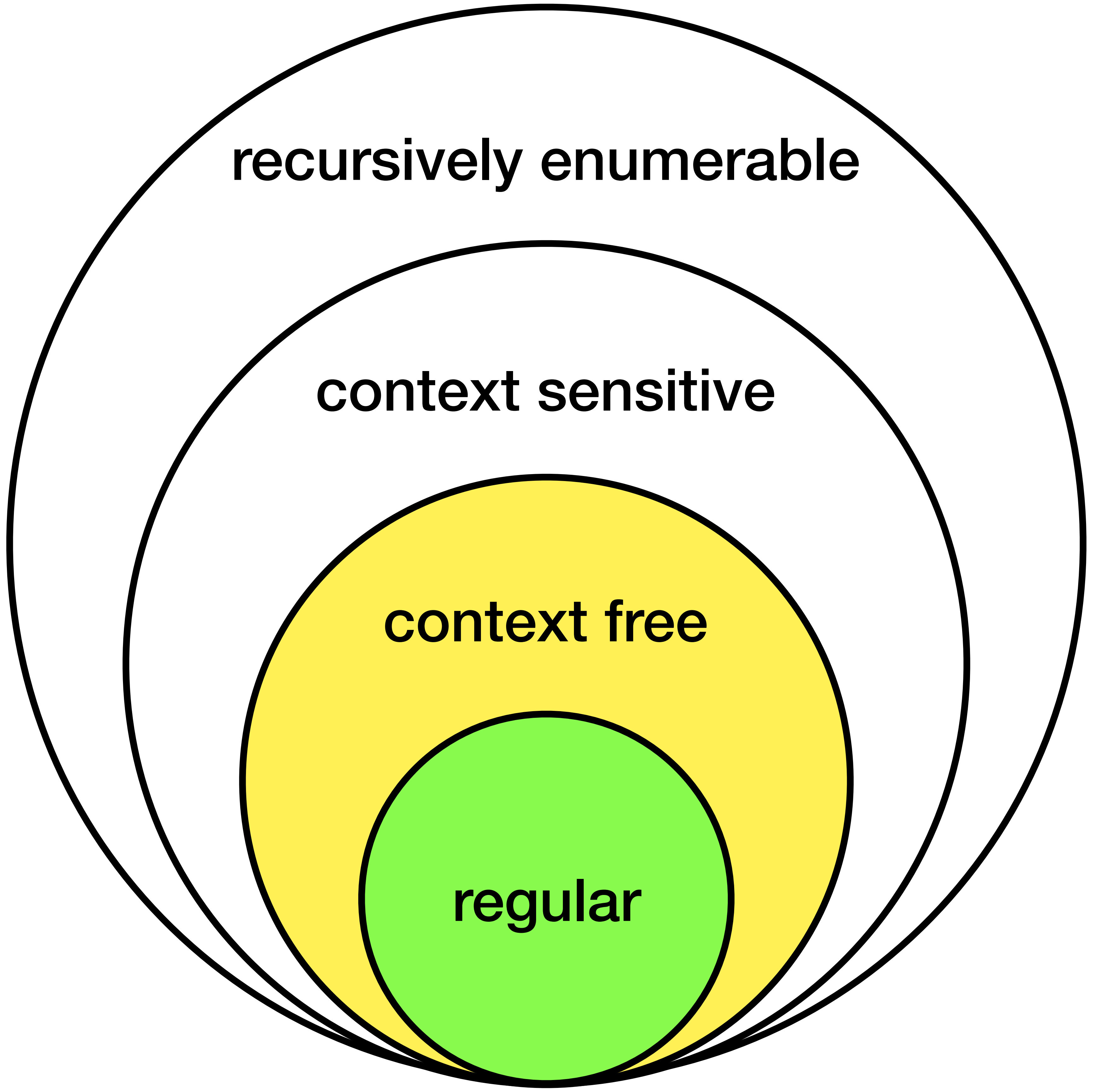
- The picture depicts regular languages as a proper subset of context-free languages.
- Thus, all regular languages are also CFLs.
  - What was the grammar that generated a regular language?



CFL/CFGs and regular languages

Recall Chomsky Heirarchy

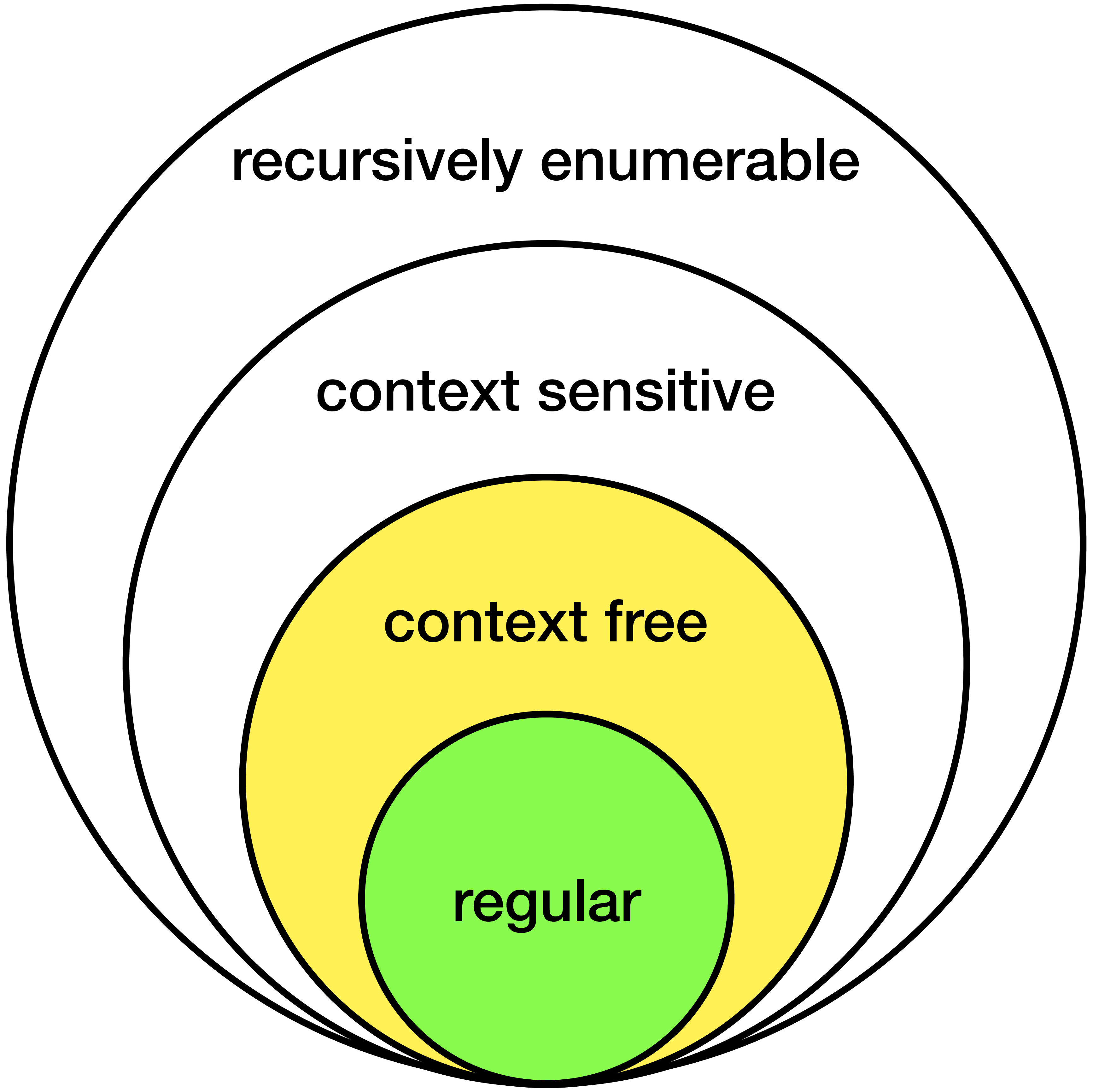
- The picture depicts regular languages as a proper subset of context-free languages.
- Thus, all regular languages are also CFLs.
  - What was the grammar that generated a regular language?
    - We can start with the DFA recognizing a regular language.



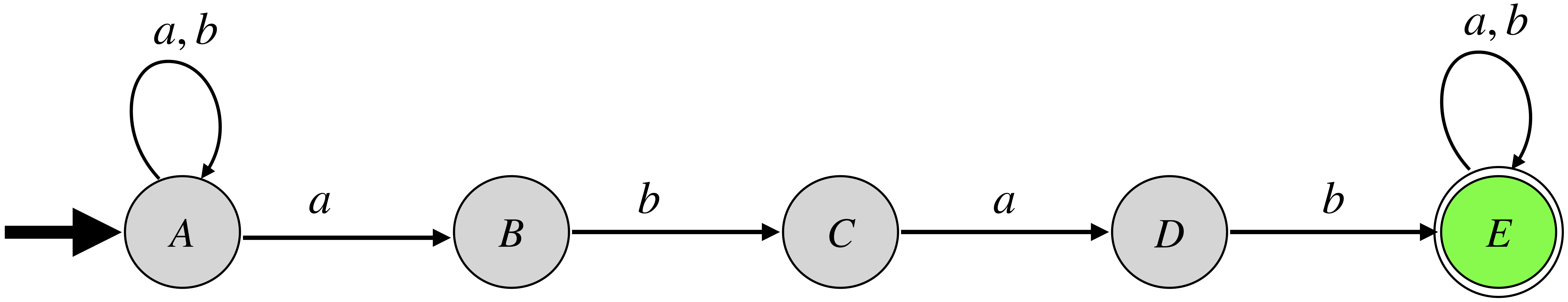
CFL/CFGs and regular languages

Recall Chomsky Heirarchy

- The picture depicts regular languages as a proper subset of context-free languages.
- Thus, all regular languages are also CFLs.
    - What was the grammar that generated a regular language?
        - We can start with the DFA recognizing a regular language.
        - Then, extend the algebraic method.



Converting DFAs into CFL

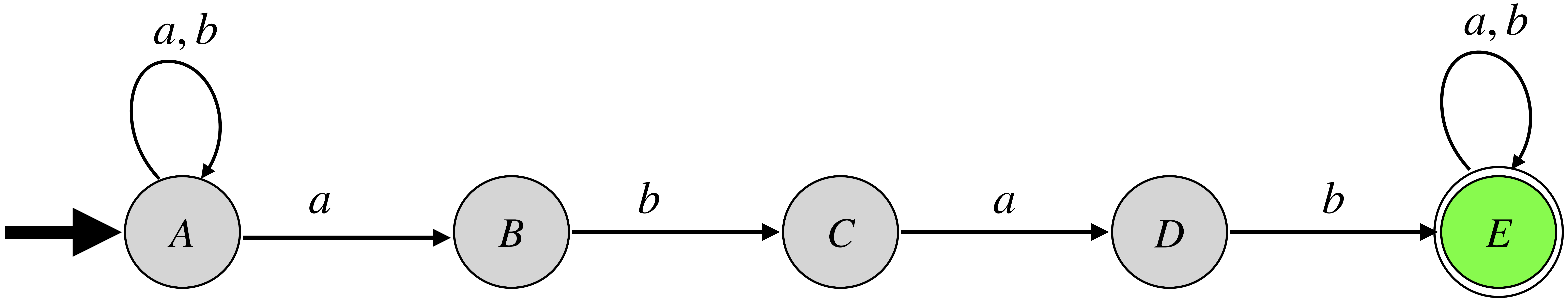


Need for a CFG

$G = (V, T, P, S)$

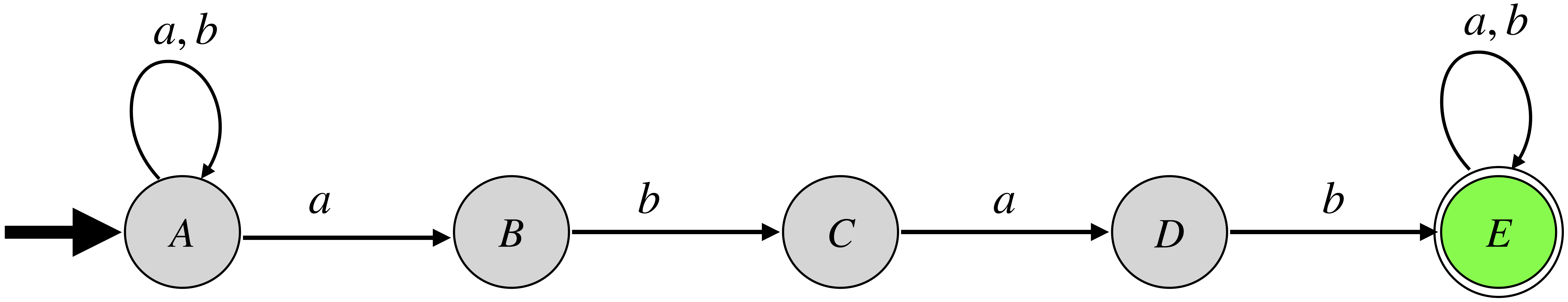
- V → variables
- T → Terminals
- P → production rules
- S → start variable

Converting regular languages into CFL



$M = (Q, \Sigma, \delta, q_0, F)$: DFA for regular language L

Converting regular languages into CFL



$M = (Q, \Sigma, \delta, q_0, F)$: DFA for regular language L

$$G = \left(\overbrace{Q}^{\text{Variables}}, \overbrace{\Sigma}^{\text{Terminals}}, \overbrace{\{q \to a\underbrace{\delta(q, a)}_{\in Q} \mid q \in Q, a \in \Sigma\} \bigcup_{q \in F} \{q \to \varepsilon\}}^{\text{Productions}}, \overbrace{q_0}^{\text{Start var}} \right)$$

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\}, \ \{a, b\} \ \left\{ \begin{array}{r} A \rightarrow aA, A \rightarrow bA, A \rightarrow aB \\ B \rightarrow bC \\ C \rightarrow aD, \\ D \rightarrow bE, \\ E \rightarrow aE, E \rightarrow bE, E \rightarrow \varepsilon \end{array} \right\}, A \right)$$

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\},\ \{a, b\}\ \left\{ \begin{array}{r} A \to aA, A \to bA, A \to aB \\ B \to bC \\ C \to aD, \\ D \to bE, \\ E \to aE, E \to bE, E \to \varepsilon \end{array} \right\}, A \right)$$

In regular languages:

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\},\ \{a, b\}\ \left\{ \begin{array}{r} A \rightarrow aA, A \rightarrow bA, A \rightarrow aB \\ B \rightarrow bC \\ C \rightarrow aD, \\ D \rightarrow bE, \\ E \rightarrow aE, E \rightarrow bE, E \rightarrow \varepsilon \end{array} \right\}, A \right)$$

In regular languages:

- Terminals can only appear on one side of the production string

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\},\ \{a, b\}\ \left\{ \begin{array}{r} A \to aA, A \to bA, A \to aB \\ B \to bC \\ C \to aD, \\ D \to bE, \\ E \to aE, E \to bE, E \to \varepsilon \end{array} \right\}, A \right)$$

In regular languages:

- Terminals can only appear on one side of the production string

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\}, \{a, b\} \left\{ \begin{array}{r} A \to aA, A \to bA, A \to aB \\ B \to bC \\ C \to aD, \\ D \to bE, \\ E \to aE, E \to bE, E \to \varepsilon \end{array} \right\}, A \right)$$

In regular languages:

- Terminals can only appear on one side of the production string
- Only one variable allowed in the production result

{0^n 1^n}   S → ε | 0S1 → problem because terminals appear on both sides

Converting regular languages into CFL

$$G = \left(\{A, B, C, D, E\},\ \{a, b\}\ \left\{ \begin{array}{r} A \to aA, A \to bA, A \to aB \\ B \to bC \\ C \to aD, \\ D \to bE, \\ E \to aE, E \to bE, E \to \varepsilon \end{array} \right\}, A \right)$$

In regular languages:

- Terminals can only appear on one side of the production string
- Only one variable allowed in the production result

Closure Properties of CFL

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_1, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Closure Properties of CFL

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_1, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Simplifying assumption: $V_1 \cap V_2 = \emptyset$, that is, non-terminals are not shared

Closure Properties of CFL

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_1, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Simplifying assumption: $V_1 \cap V_2 = \emptyset$, that is, non-terminals are not shared

- CFLs are closed under union: $L_1 \cup L_2$ is a CFL.

Closure Properties of CFL

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_1, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Simplifying assumption: $V_1 \cap V_2 = \varnothing$, that is, non-terminals are not shared

- CFLs are closed under union: $L_1 \cup L_2$ is a CFL.
- CFLs are closed under concatenation: $L_1 \cdot L_2$ is a CFL.

Closure Properties of CFL

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_1, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Simplifying assumption: $V_1 \cap V_2 = \varnothing$, that is, non-terminals are not shared

- CFLs are closed under union: $L_1 \cup L_2$ is a CFL.
- CFLs are closed under concatenation: $L_1 \cdot L_2$ is a CFL.
- CFLs are closed under Kleene star: L_k CFL implies L_k^* is a CFL.

NOT CLOSED

→ intersection

→ complement

Closure Properties of CFL

Closure under concatenation

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ be CFGs for $L_1 = L(G_1)$ and $L_2 = L(G_2)$. Suppose $L = L_1 \cdot L_2$. What is a grammar for L?

Simplifying assumption: $V_1 \cap V_2 = \emptyset$, that is, non-terminals are not shared.

$$G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$$

$$S \to S_1 S_2$$

$$P = P_1 \cup P_2 \cup \{S \to S_1 S_2\}$$

Closure Properties of CFL

Closure under Kleene star

Let $G_1 = (V_1, T_1, P_1, S_1)$ be CFG for $L_1 = L(G_1)$. Suppose $L = L_1^*$. What is a grammar for L?

Left as an exercise

Pushdown automata

Pushdown automata

The machine that recognizes CFGs

We established that $\{0^n 1^n \mid n \geq 0\}$ is a CFL but not a regular language.

Pushdown automata

The machine that recognizes CFGs

We established that $\{0^n 1^n \mid n \geq 0\}$ is a CFL but not a regular language.

We have NFAs from regular languages. What can we add to enable them to recognize CFLs?

Pushdown automata

The machine that recognizes CFGs

We established that $\{0^n 1^n \,|\, n \geq 0\}$ is a CFL but not a regular language.

We have NFAs from regular languages. What can we add to enable them to recognize CFLs?

The key idea is that CFGs allow recursive definitions.

Pushdown automata

The machine that recognizes CFGs

We established that $\{0^n 1^n \mid n \geq 0\}$ is a CFL but not a regular language.

We have NFAs from regular languages. What can we add to enable them to recognize CFLs?

The key idea is that CFGs allow recursive definitions.

(ECE 220) What enables recursion in programming?

Pushdown automata

The machine that recognizes CFGs

We established that $\{0^n1^n \,|\, n \geq 0\}$ is a CFL but not a regular language.

We have NFAs from regular languages. What can we add to enable them to recognize CFLs?

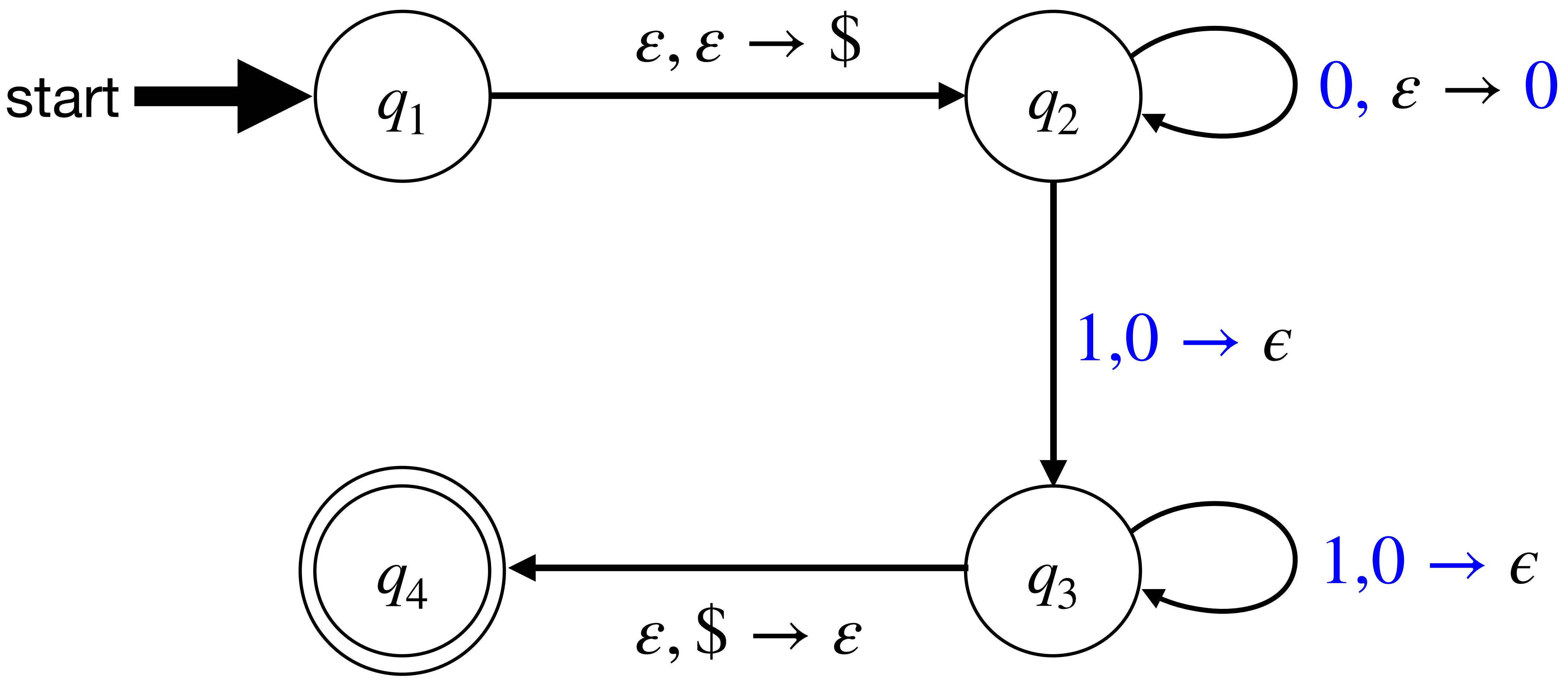
The key idea is that CFGs allow recursive definitions.

(ECE 220) What enables recursion in programming?

We need a stack!

Push-down Automata

The machine that generates CFGs



Each transition is formatted as:

<token read>, <stack pop> → <stack push>

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a 6-tuple where

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a 6-tuple where

- Q is a finite set whose elements are called states,

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = \left(Q, \Sigma, \Gamma, \delta, s, A\right)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,

together captures state (Q) and stack configuration

- $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\} \to \mathscr{P}\left(Q \times (\Gamma \cup \{\varepsilon\})\right)$ is the transition function

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = \left(Q, \Sigma, \Gamma, \delta, s, A\right)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\} \to \mathscr{P}\left(Q \times (\Gamma \cup \{\varepsilon\})\right)$ is the transition function
- s is the start state

Formal Tuple Notation

Definition: A non-deterministic push-down automaton $P = \left(Q, \Sigma, \Gamma, \delta, s, A\right)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\} \to \mathscr{P}\left(Q \times (\Gamma \cup \{\varepsilon\})\right)$ is the transition function
- s is the start state
- A is the set of accepting states

Formal Tuple Notation

Non-deterministic PDAs are more "powerful" than deterministic PDAs. Hence, we'll only be talking about non-deterministic PDAs.

NPDAs can recognize more than DPDAs.

Definition: A non-deterministic push-down automaton $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a 6-tuple where

- Q is a finite set whose elements are called states,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\} \rightarrow \mathscr{P}\left(Q \times (\Gamma \cup \{\varepsilon\})\right)$ is the transition function
- s is the start state
- A is the set of accepting states

CFGs and PDAs

Convert a CFG to a PDA

Consider,

$$S \rightarrow 0S \mid 1 \mid \epsilon$$

What is a PDA for this?

CFGs and PDAs

Convert a CFG to a PDA

Consider,

$$S \rightarrow 0S \mid 1 \mid \epsilon$$

What is a PDA for this?

CFGs and PDAs

Convert a CFG to a PDA

Consider,

$$S \to 0S \mid 1 \mid \epsilon$$

What is a PDA for this?

Key idea: Recreate the string on the stack

CFGs and PDAs

Convert a CFG to a PDA

Consider,

$$S \rightarrow 0S \mid 1 \mid \epsilon$$

What is a PDA for this?

Key idea: Recreate the string on the stack

- Every time we see a non-terminal, we replace it with one of the replacement rules.

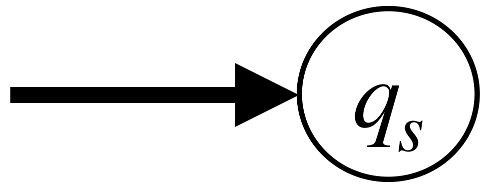
- Every time we see a terminal symbol, we take that symbol from the input.

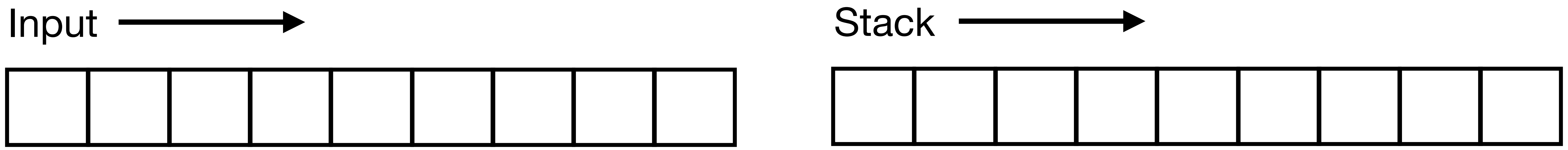
- If we reach a point where the stack and input are empty, then we accept the string.

CFGs and PDAs

Convert a CFG to a PDA

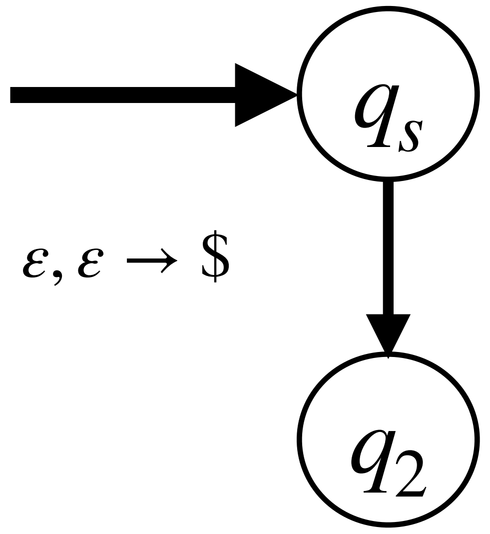
$$S \to 0S \ | \ 1 \ | \epsilon$$





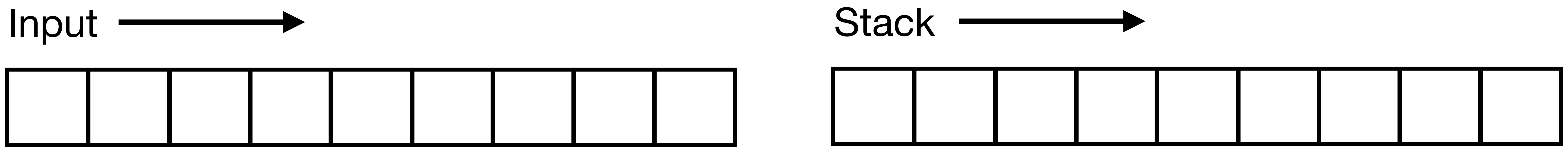
CFGs and PDAs

Convert a CFG to a PDA



$$S \to 0S \mid 1 \mid \epsilon$$

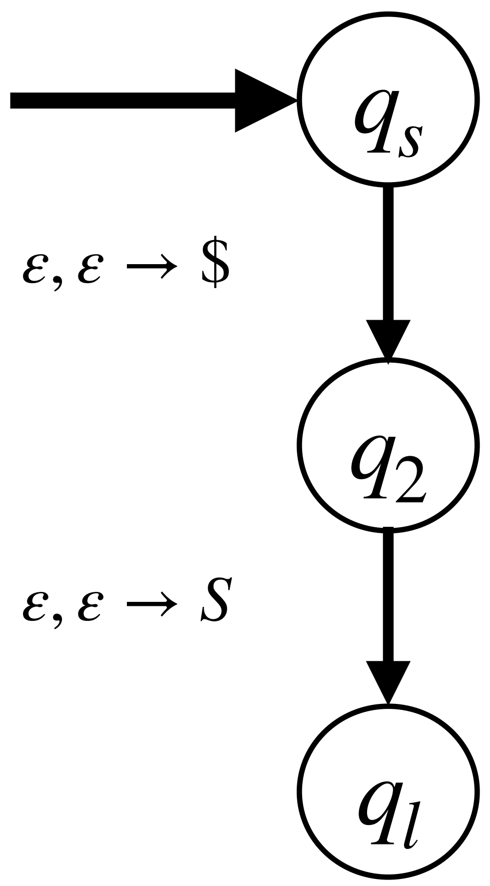
- First let's put in a $\$$ to mark the end of the string



input token, pop item → push item

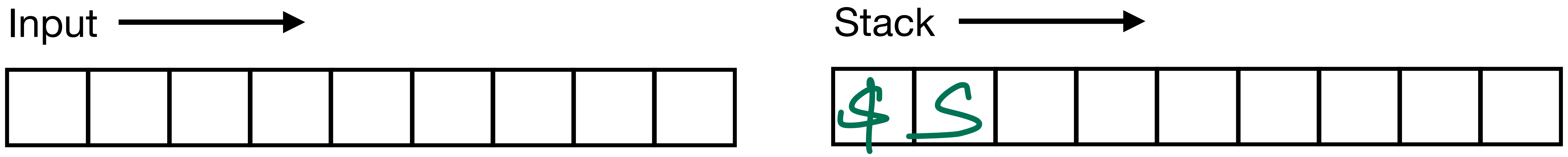
CFGs and PDAs

Convert a CFG to a PDA



$$S \to 0S \mid 1 \mid \epsilon$$

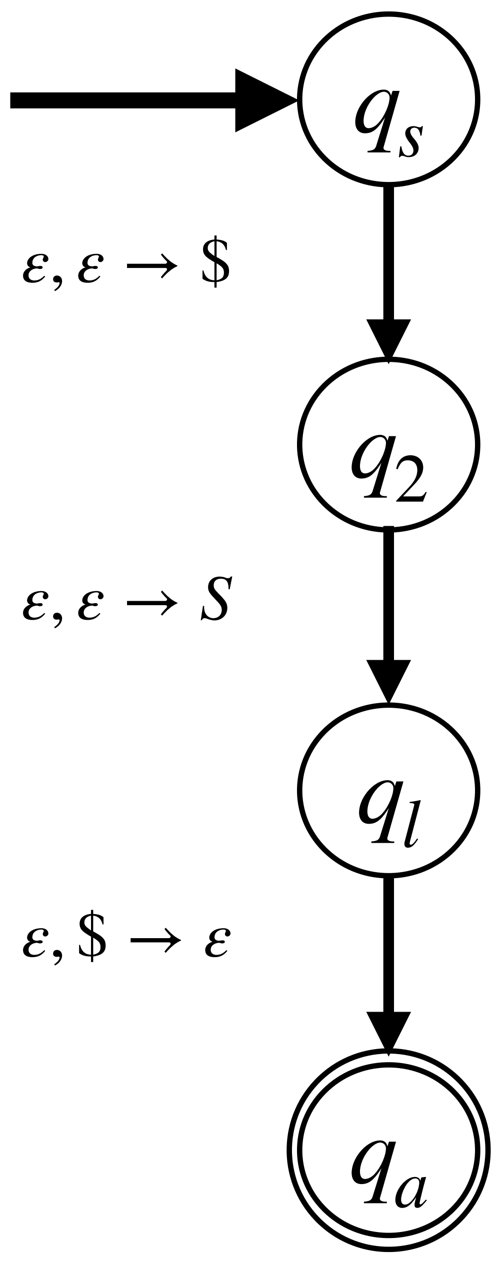
- First let's put in a $\$$ to mark the end of the string
- Also let's put in the start symbol on the stack.



CFGs and PDAs

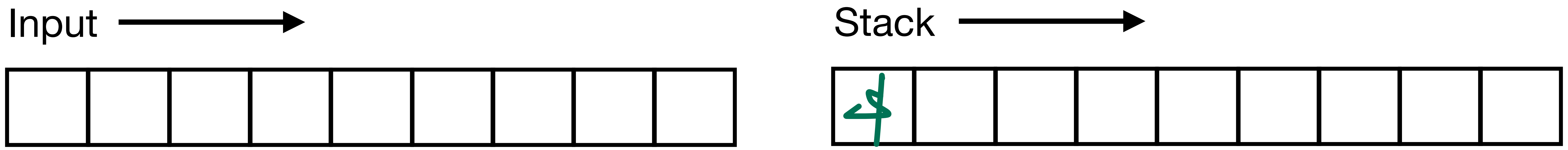
Convert a CFG to a PDA

$S \to 0S \mid 1 \mid \epsilon$



Next we want to add a loop for every non-terminal symbol that replaces that non-terminal with the result.

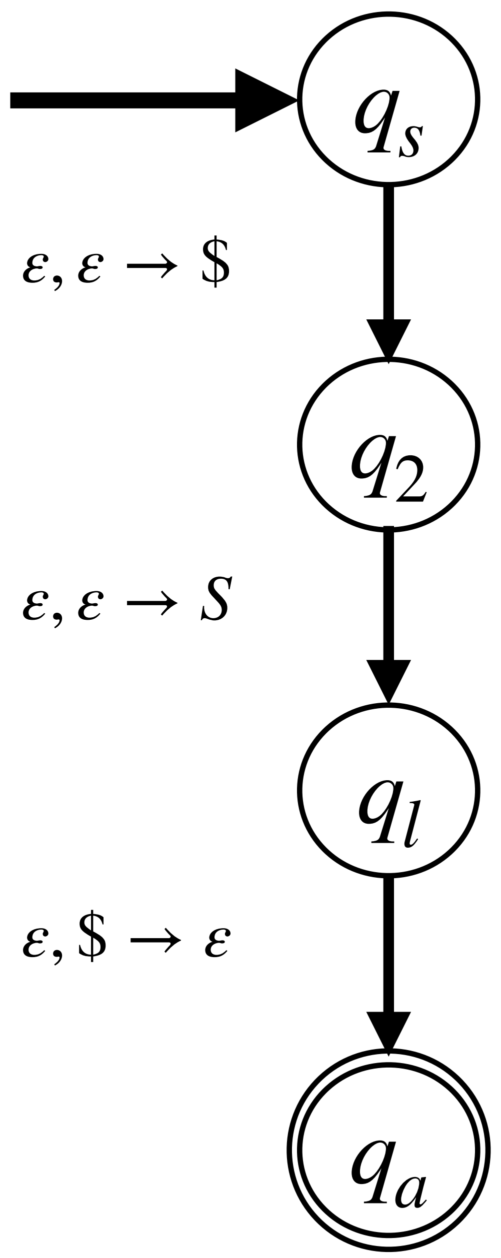
Consider the rule: $S \to 0S$



CFGs and PDAs

Convert a CFG to a PDA

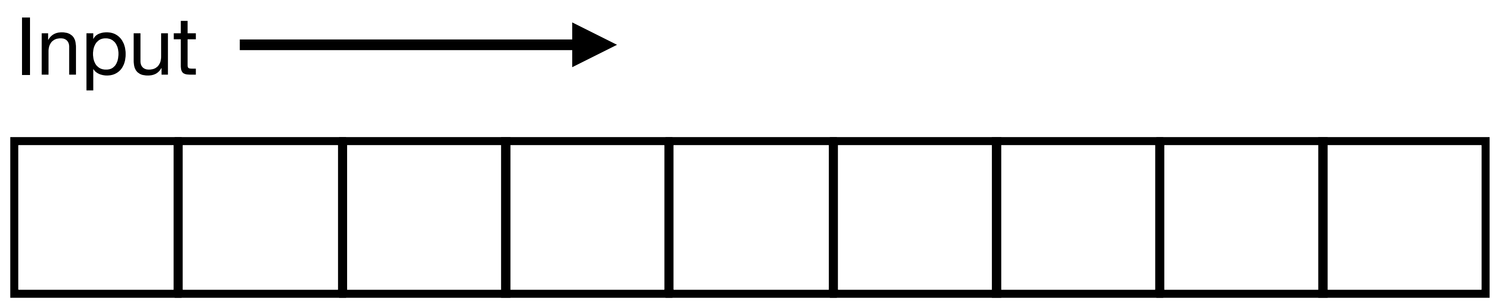
$S \rightarrow 0S \mid 1 \mid \epsilon$

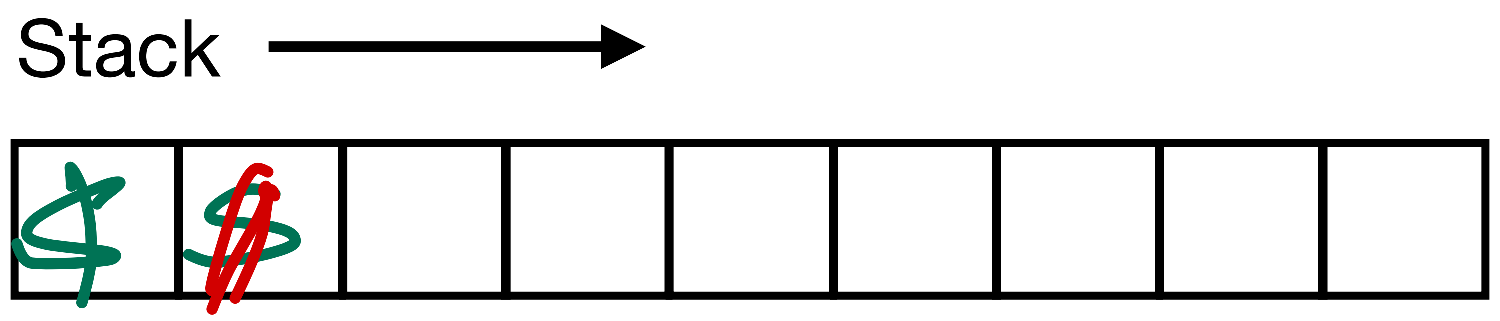


Next we want to add a loop for every non-terminal symbol that replaces that non-terminal with the result.

Consider the rule: $S \rightarrow 0S$

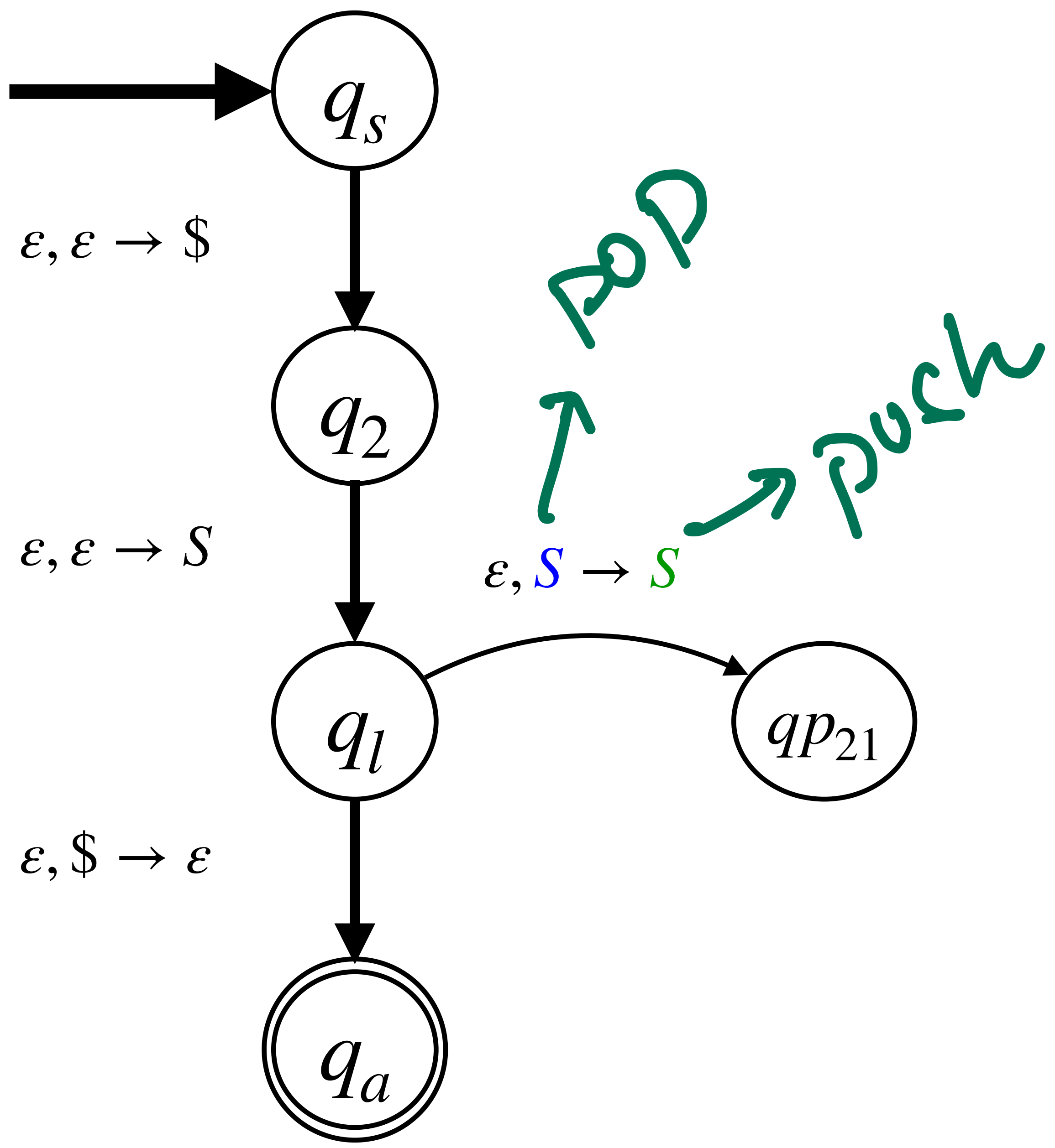
- So we got to pop S the non-terminal and …





CFGs and PDAs

Convert a CFG to a PDA

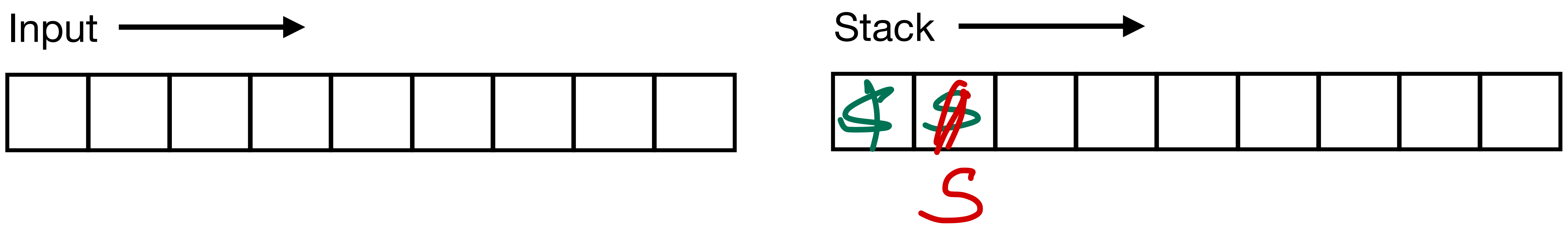


$S \to 0S \mid 1 \mid \epsilon$

Next we want to add a loop for every non-terminal symbol that replaces that non-terminal with the result.

Consider the rule: $S \to 0S$

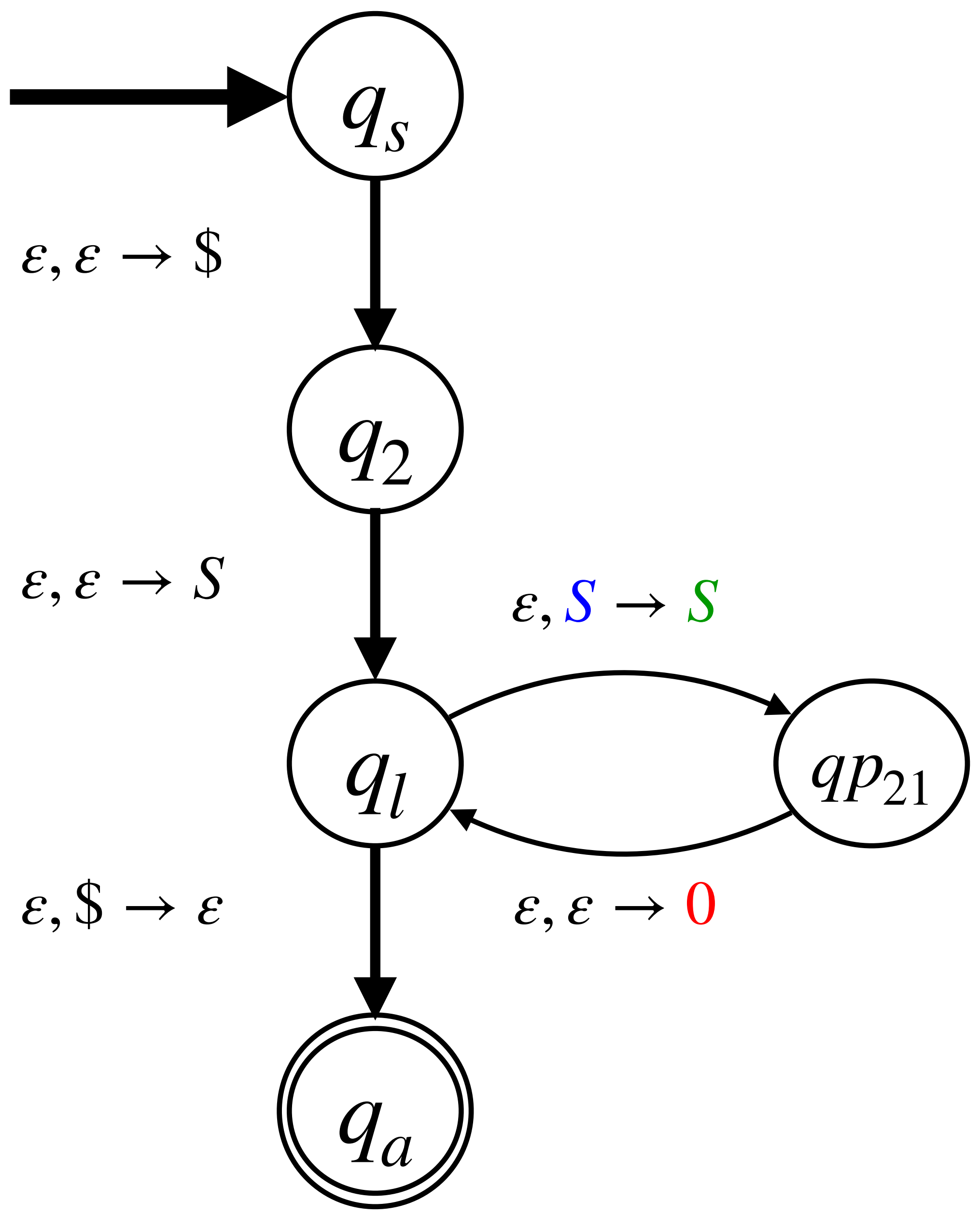
- So we got to pop S the non-terminal and …
- Add a non-terminal S to the stack.



CFGs and PDAs

Convert a CFG to a PDA

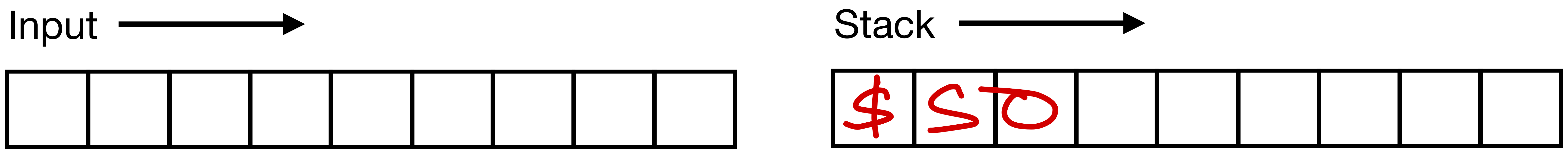
$S \rightarrow 0S \ | \ 1 \ | \ \epsilon$



Next we want to add a loop for every non-terminal symbol that replaces that non-terminal with the result.

Consider the rule: $S \rightarrow 0S$

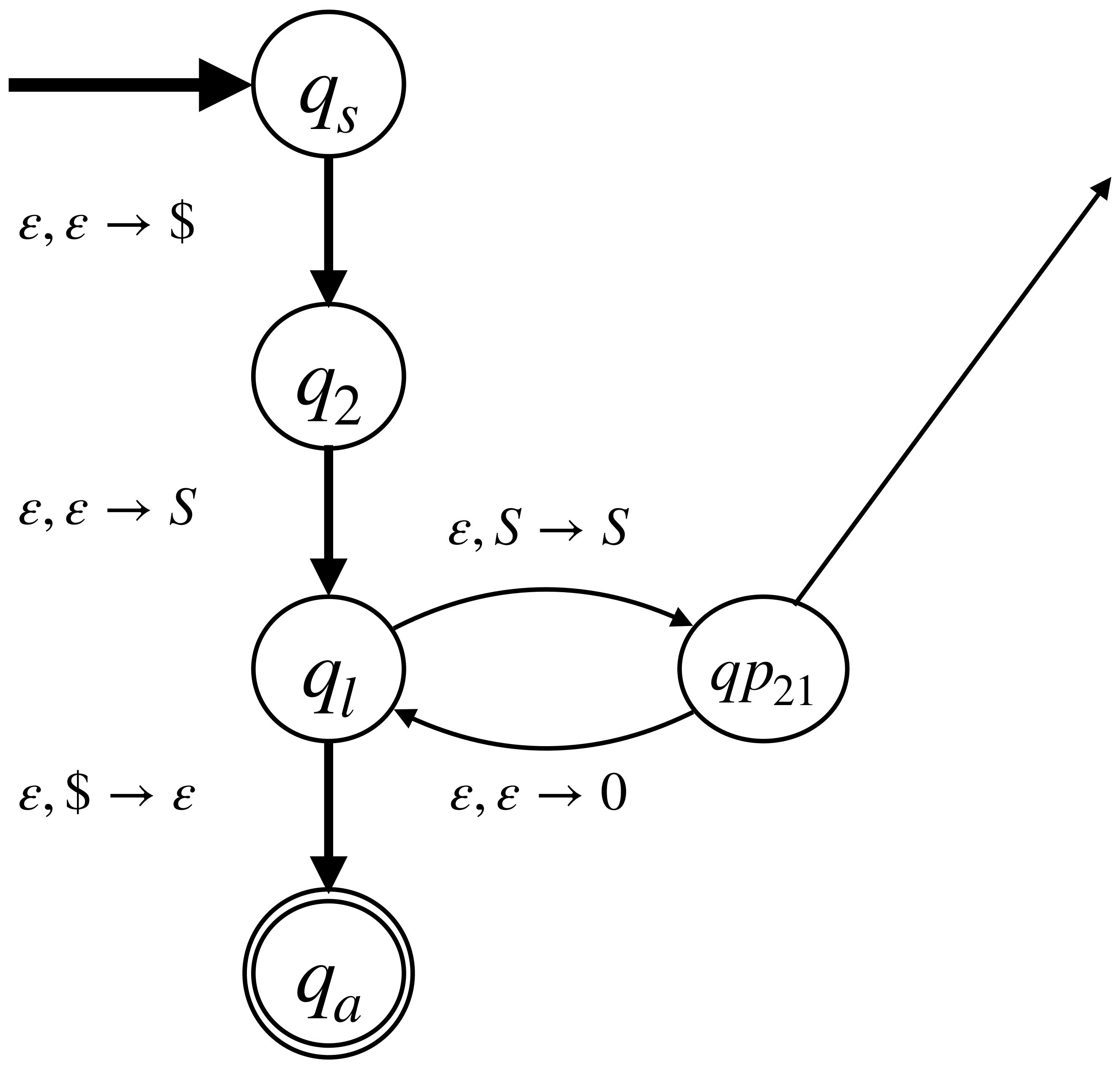
- So we got to pop S the non-terminal and …
- Add a non-terminal S to the stack.
- And add a terminal 0 to the stack.



CFGs and PDAs

Convert a CFG to a PDA

$S \to 0S \mid 1 \mid \epsilon$



Is this state necessary?

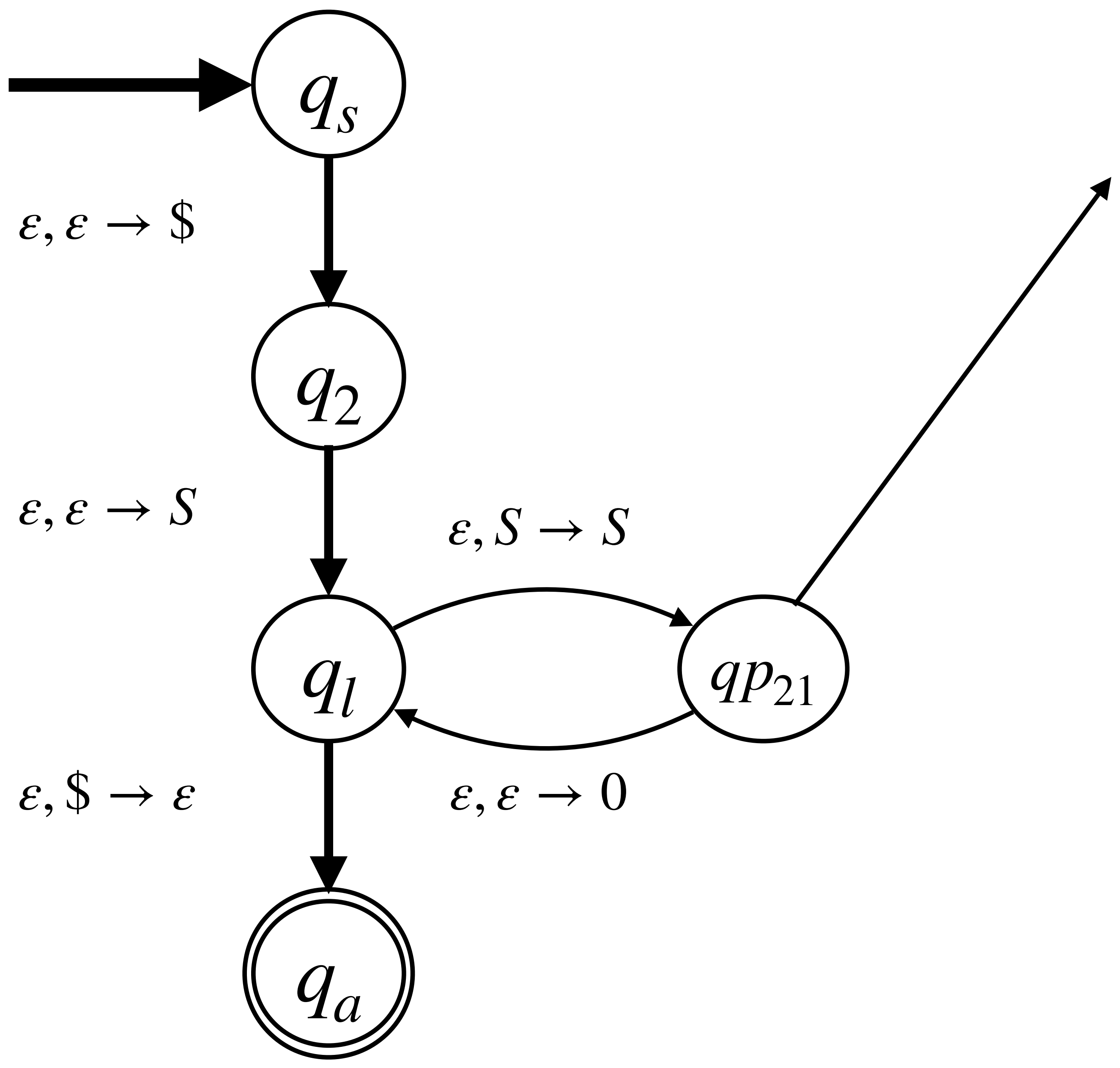
Input

Stack

CFGs and PDAs

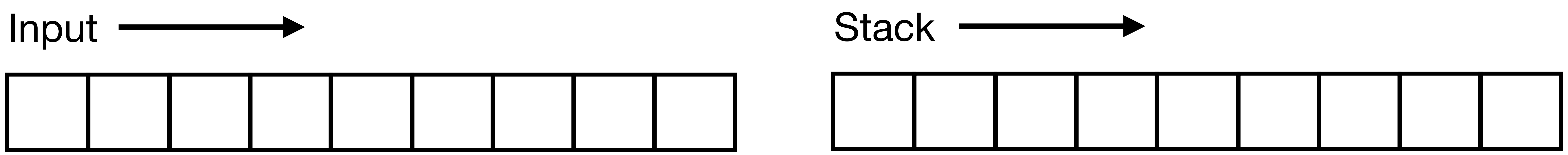
Convert a CFG to a PDA

$S \rightarrow 0S \mid 1 \mid \epsilon$



Is this state necessary?

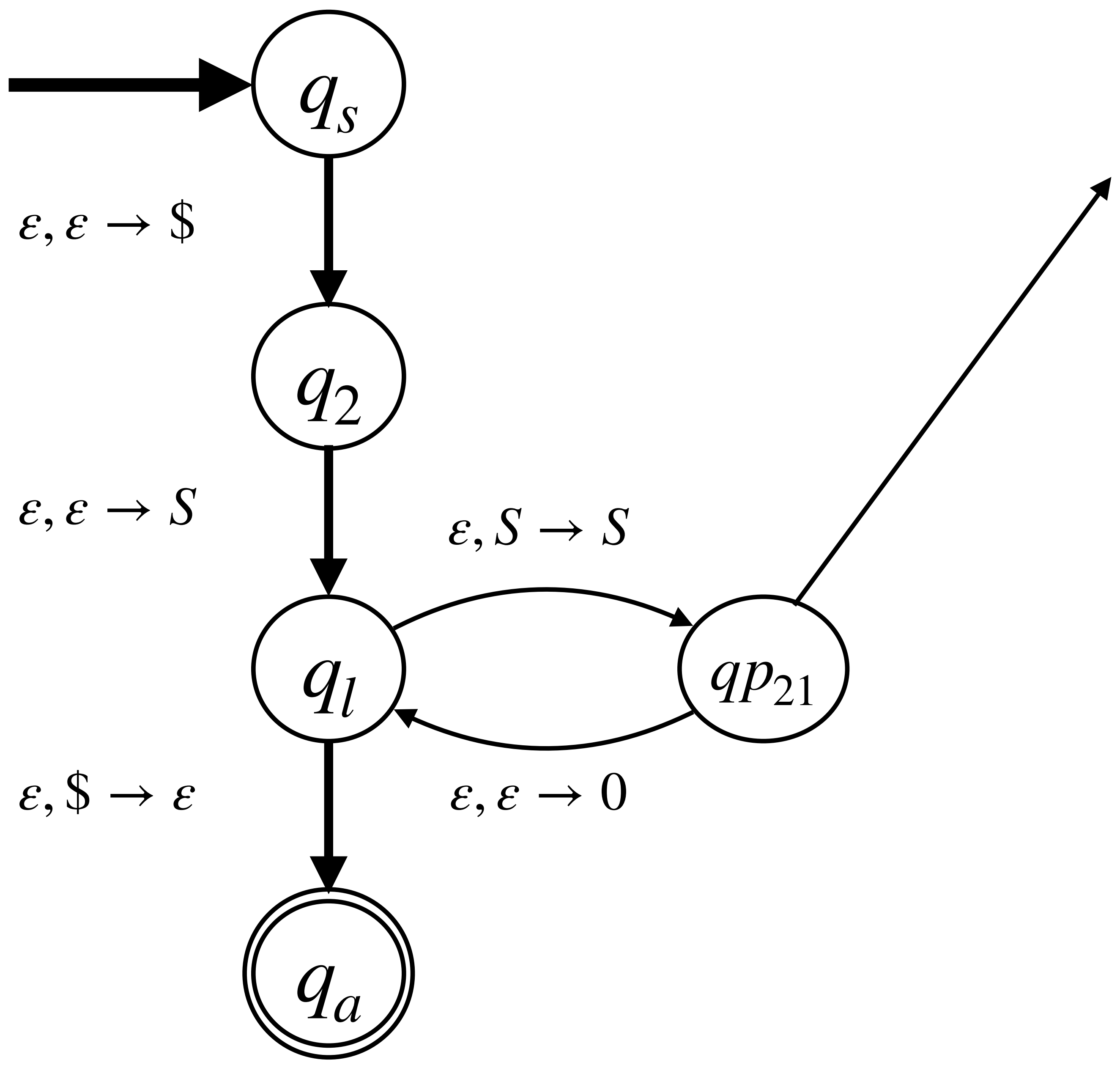
- Recall generalized NFAs?



CFGs and PDAs

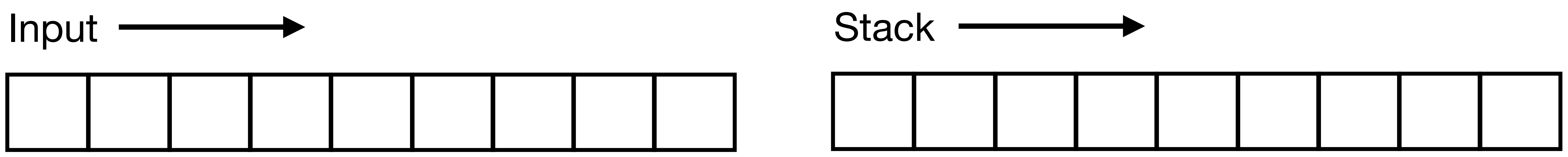
Convert a CFG to a PDA

$S \rightarrow 0S \mid 1 \mid \epsilon$



Is this state necessary?

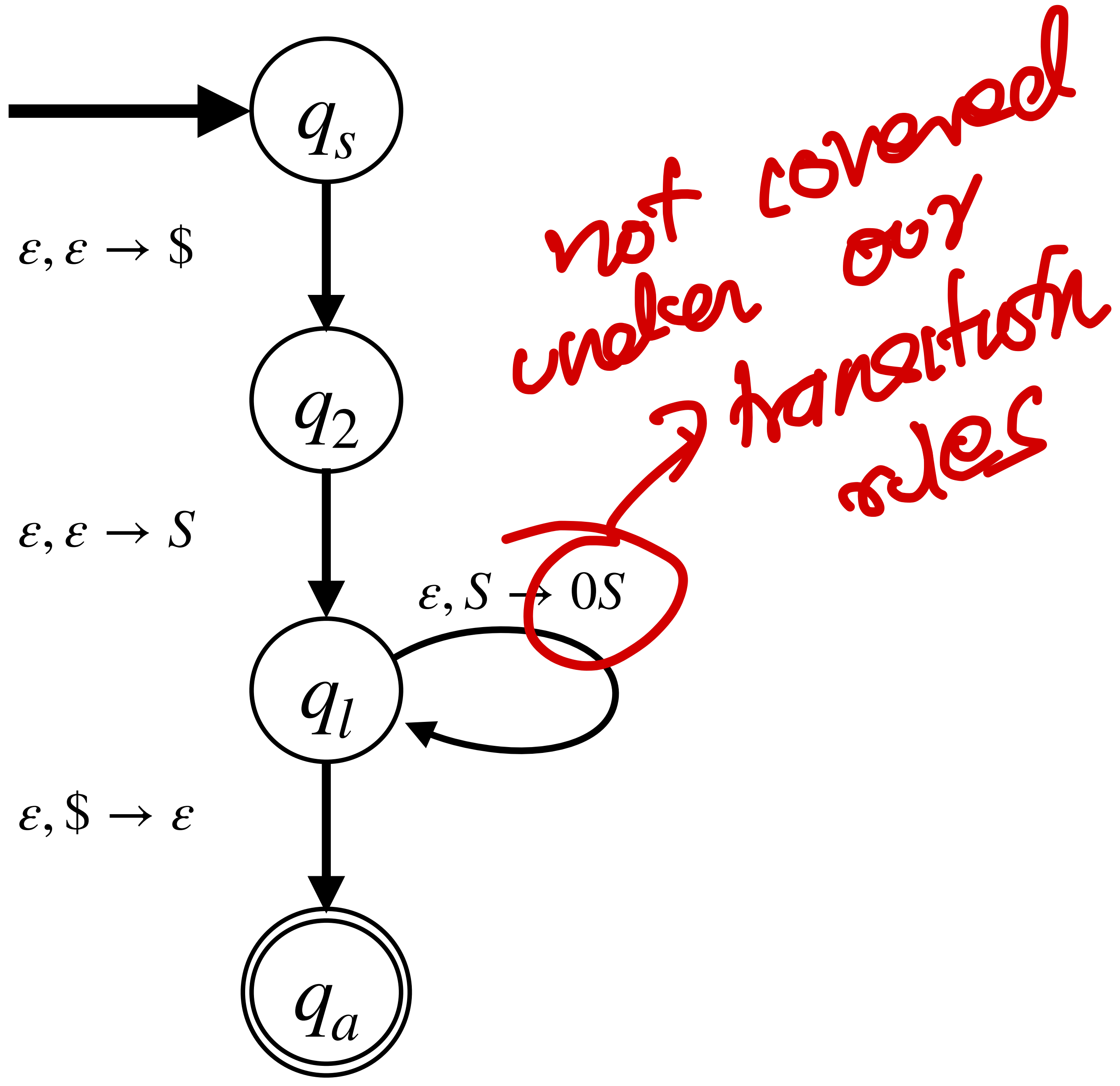
- Recall generalized NFAs?
- Can follow same route to allow entire strings to be pushed onto stack.



CFGs and PDAs

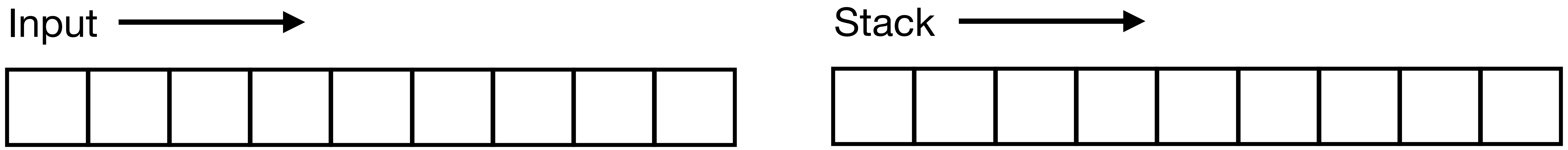
Convert a CFG to a PDA

$$S \rightarrow 0S \mid 1 \mid \epsilon$$



Is this state necessary?

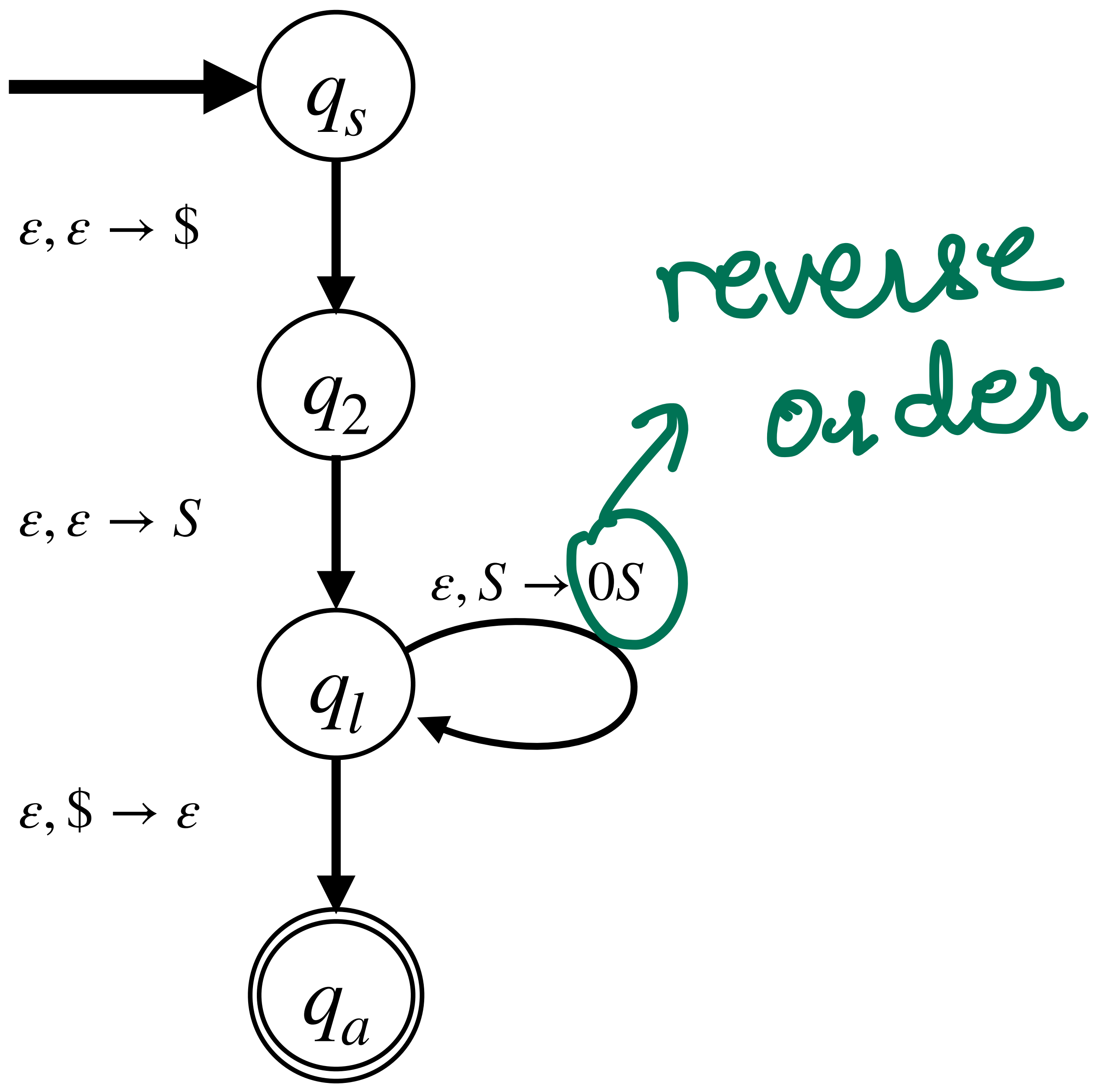
- Recall generalized NFAs?
- Can follow same route to allow entire strings to be pushed onto stack.



CFGs and PDAs

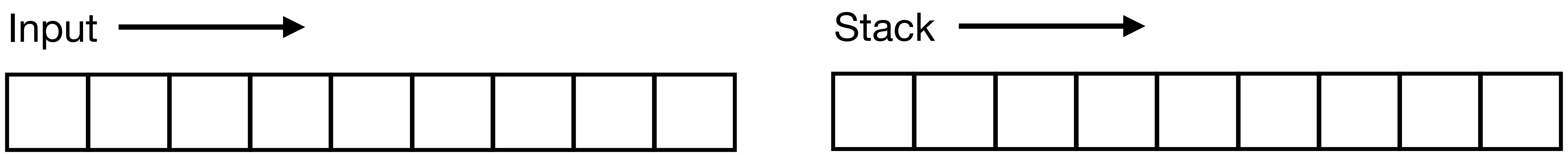
Convert a CFG to a PDA

$$S \rightarrow 0S \mid 1 \mid \epsilon$$



Is this state necessary?

- Recall generalized NFAs?
- Can follow same route to allow entire strings to be pushed onto stack.
- But we are going to stick with PDAs.

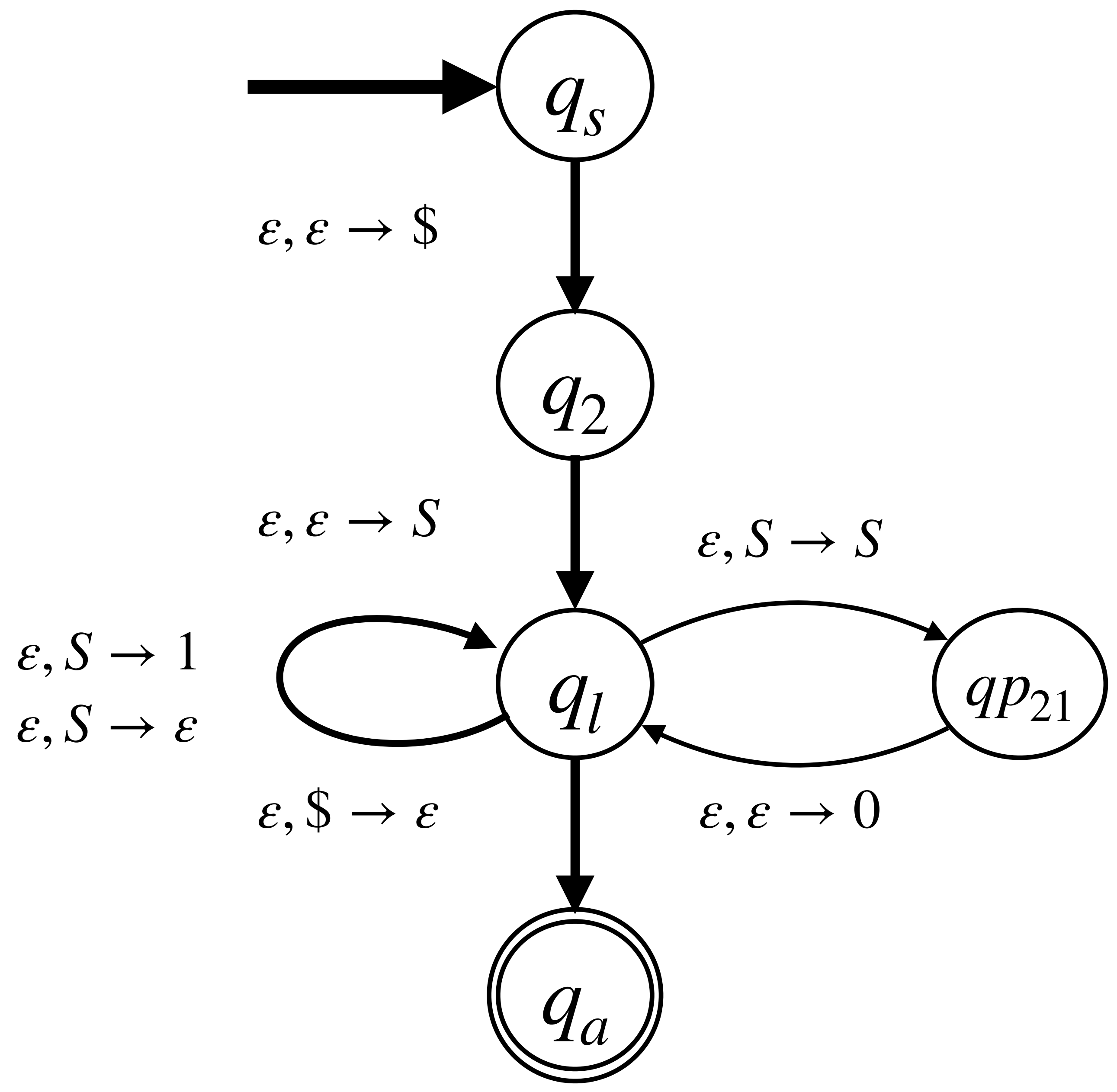


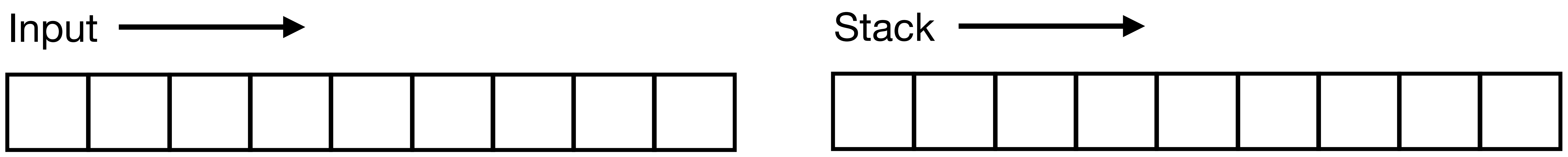
CFGs and PDAs

Convert a CFG to a PDA

$$S \to 0S \;|\; 1 \;|\; \varepsilon$$

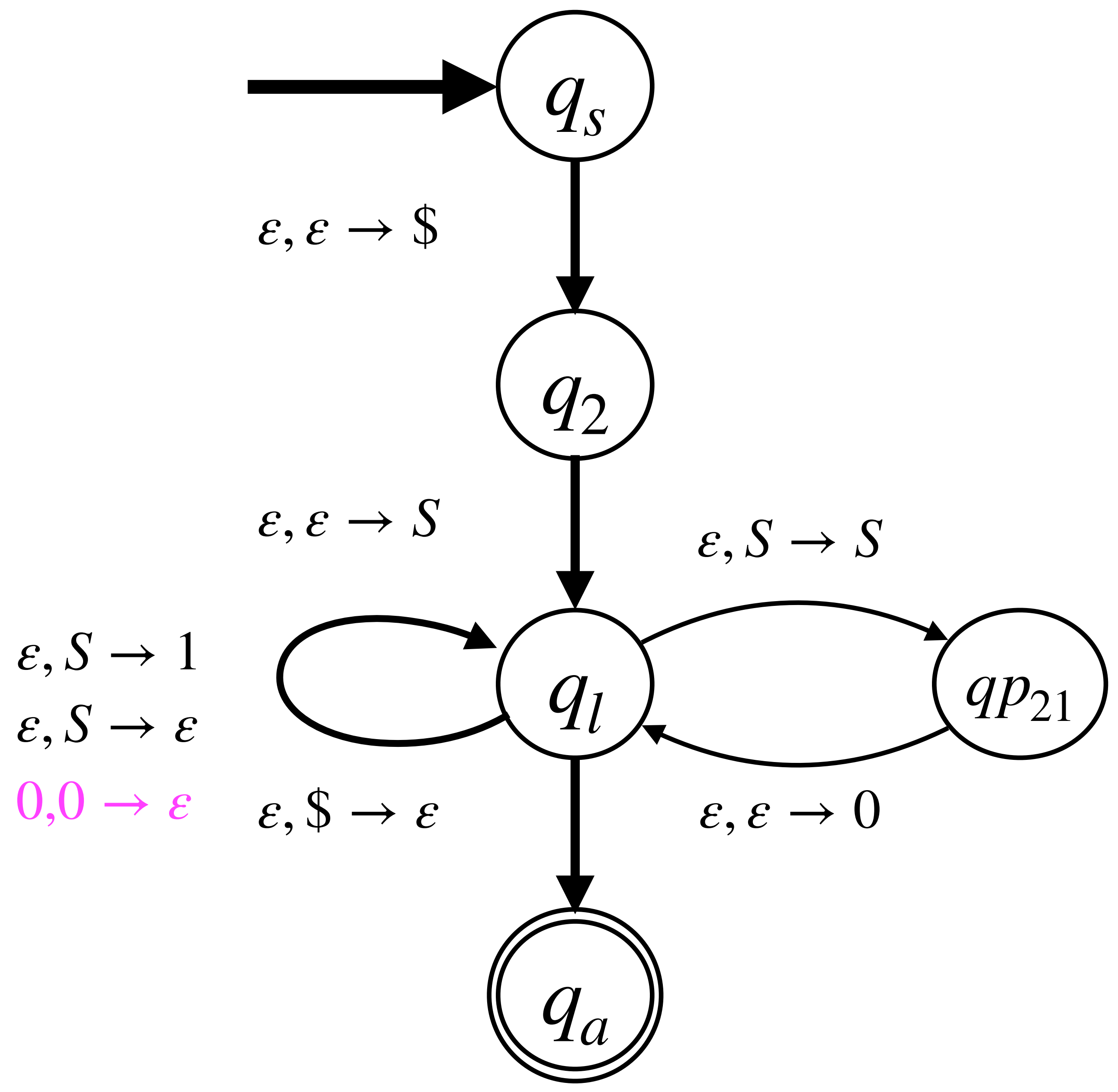
- If we see a ~~non~~-terminal symbol on the stack, then we can cross that symbol from the input.





CFGs and PDAs

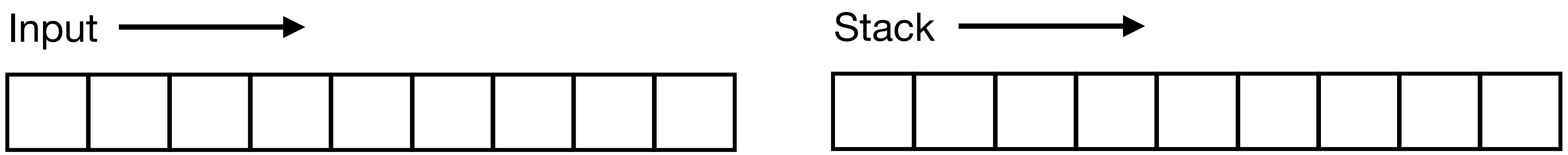
Convert a CFG to a PDA



$$S \to 0S \mid 1 \mid \varepsilon$$

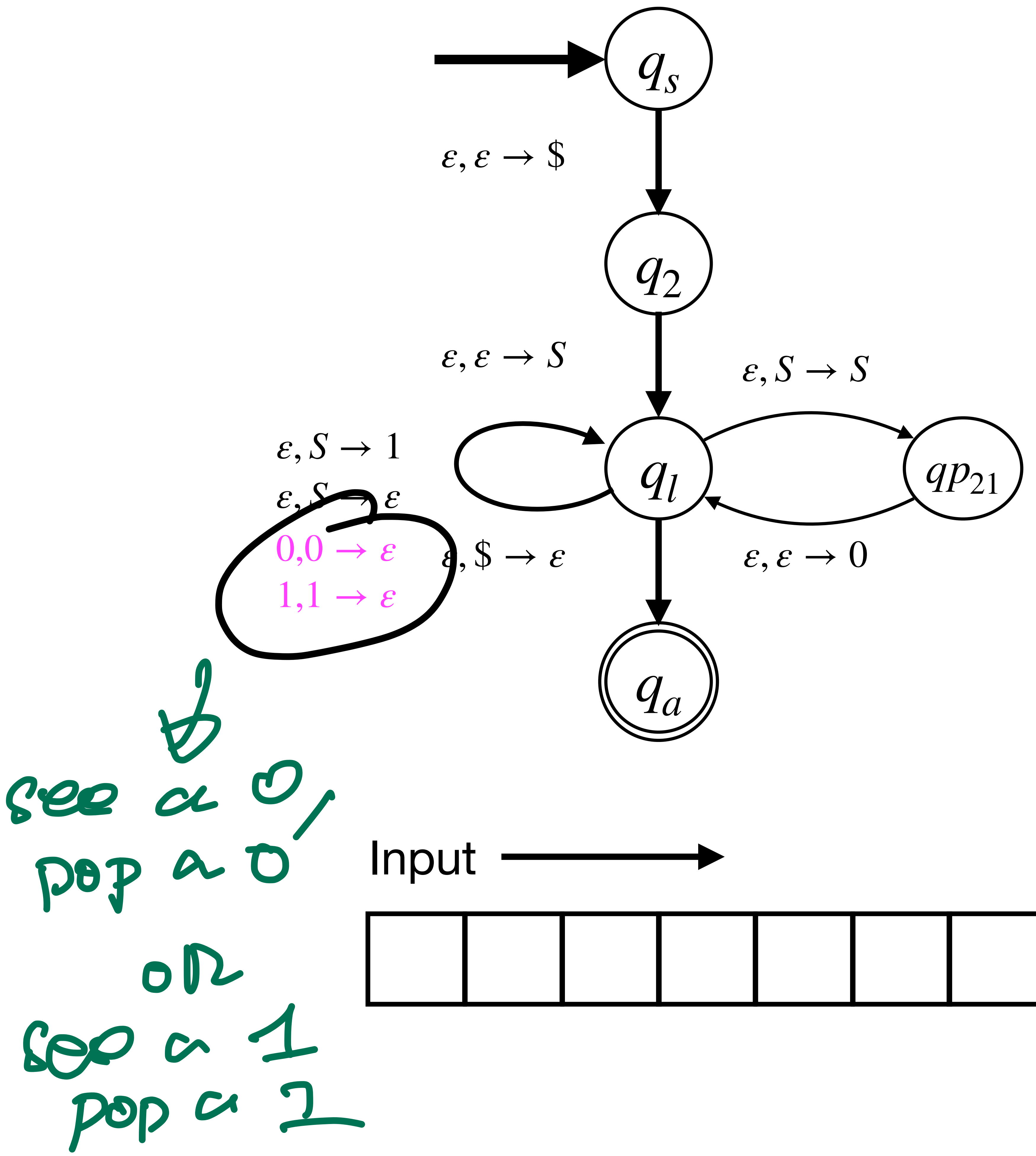
typo

- If we see a ~~non~~-terminal symbol on the stack, then we can cross that symbol from the input.



CFGs and PDAs

Convert a CFG to a PDA



$$S \to 0S \mid 1 \mid \varepsilon$$

- If we see a ~~non~~-terminal symbol on the stack, then we can cross that symbol from the input.

Input →

Stack →

Example from Kani Archive

Start here next lecture.

Convert a CFG to a PDA

Another example