COMP-330
Theory of Computation

Fall 2019 -- Prof. Claude Crépeau

Lec. 12 :
PDA-CFG equivalence

MORE Midterm INFO

Posted Oct 9, 2019 3:57 PM

The midterm exam will take place on the evening of Oct 17th from 18:00 to 19:30.

Location: Montreal Neurological Institute Jeanne Timmins Amphitheatre
3801 University St.

Practice (2017) midterm exam HERE.

As indicated on the 2017 exam above, my exams are always OPEN BOOK.

All documentation is always permitted.

MORE Midterm INFO

Posted Oct 9, 2019 3:57 PM

The midterm exam will take place on the evening of Oct 17th from 18:00 to 19:30.

Location: Montreal Neurological Institute Jeanne Timmins Amphitheatre
3801 University St.

Practice (2017) midterm exam HERE.

As indicated on the 2017 exam above, my exams are always OPEN BOOK.

All documentation is always permitted.

send me e-mail for conflicts
N O W !!!

MORE Midterm INFO

Posted Oct 9, 2019 3:57 PM

The midterm exam will take place on the evening of Oct 17th from 18:00 to 19:30.

Location: Montreal Neurological Institute Jeanne Timmins Amphitheatre
3801 University St.

Practice (2017) midterm exam HERE.

As indicated on the 2017 exam above, my exams are always OPEN BOOK.

All documentation is always permitted.

send me e-mail for conflicts
. N O W !!! .

Announcements

HW2 details

Posted Oct 9, 2019 9:44 PM

QUESTION 1.998

The alphabet for parts a.--f. is $\Sigma=\{0,1,2,3,4,5,6,7,8,9\}$.

d) Exceptionally you may provide an NFA instead of a regular expression if the language is actually regular. ;-)

e) You may use the following lemma (without proving it) to prove this one:

For every $\epsilon > 0$, there exists integers a and b such that $1 < 2^a/5^b < 1+\epsilon$.

HINT: Use the Myhill-Nerode Theorem instead of the pumping lemma...

PDA vs CFG

THEOREM 2.20

A language is context free *if and only if* some pushdown automaton recognizes it.

PDA vs CFG

THEOREM 2.20

A language is context free *if and only if* some pushdown automaton recognizes it.

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

PDA vs CFG

THEOREM 2.20

A language is context free *if and only if* some pushdown automaton recognizes it.

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

CFG to PDA

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

CFG to PDA

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

The following is an informal description of P.

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

EXAMPLE of CFG

EXAMPLE 2.4

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\texttt{a}, \texttt{+}, \texttt{x}, \texttt{(}, \texttt{)}\}$. The rules are

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle \texttt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \texttt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a} \end{aligned}$$

⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol \$ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $
- (a+a)xa Stack: ⟨EXPR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $
- (a+a)xa Stack: ⟨EXPR⟩$
- (a+a)xa Stack: ⟨TERM⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol \$ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: \$
- (a+a)xa Stack: ⟨EXPR⟩\$
- (a+a)xa Stack: ⟨TERM⟩\$
- (a+a)xa Stack: ⟨TERM⟩x⟨FACTOR⟩\$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $
- (a+a)xa Stack: ⟨EXPR⟩$
- (a+a)xa Stack: ⟨TERM⟩$
- (a+a)xa Stack: ⟨TERM⟩x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $
- (a+a)xa Stack: ⟨EXPR⟩$
- (a+a)xa Stack: ⟨TERM⟩$
- (a+a)xa Stack: ⟨TERM⟩x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩x⟨FACTOR⟩$
- (a+a)xa Stack: (⟨EXPR⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack:
- (a+a)xa Stack: $
- (a+a)xa Stack: ⟨EXPR⟩$
- (a+a)xa Stack: ⟨TERM⟩$
- (a+a)xa Stack: ⟨TERM⟩x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩x⟨FACTOR⟩$
- (a+a)xa Stack: (⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩+⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a+⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: +⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: +⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨EXPR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨EXPR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a+⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: +⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨TERM⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$
- (a+a)xa Stack:)x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$
- (a+a)xa Stack:)x⟨FACTOR⟩$
- (a+a)xa Stack: x⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$
- (a+a)xa Stack:)x⟨FACTOR⟩$
- (a+a)xa Stack: x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$
- (a+a)xa Stack:)x⟨FACTOR⟩$
- (a+a)xa Stack: x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩$
- (a+a)xa Stack: a$

- ⟨EXPR⟩ → ⟨EXPR⟩+⟨TERM⟩ | ⟨TERM⟩
  ⟨TERM⟩ → ⟨TERM⟩x⟨FACTOR⟩ | ⟨FACTOR⟩
  ⟨FACTOR⟩ → (⟨EXPR⟩) | a

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps forever.

a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.

c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

- (a+a)xa Stack: ⟨FACTOR⟩)x⟨FACTOR⟩$
- (a+a)xa Stack: a)x⟨FACTOR⟩$
- (a+a)xa Stack:)x⟨FACTOR⟩$
- (a+a)xa Stack: x⟨FACTOR⟩$
- (a+a)xa Stack: ⟨FACTOR⟩$
- (a+a)xa Stack: a$
- (a+a)xa Stack: $

CFG to PDA

- Proof: Given a CFG $G=(V,\Sigma,R,S)$, we now construct a PDA $P=(Q,\Sigma,\Gamma,\delta,q_0,F)$ for it.

- We define a special notation to write an entire string on the stack in one step.
- We can simulate this action by adding extra states to write the string one symbol at a time.

CFG to PDA

- Let q and r be states of the PDA and let $a \in \Sigma_\varepsilon$ $s \in \Gamma_\varepsilon$.

- Starting in state q, say we want to read a from the input and pop s from the stack. Moreover we want to push string $u=u_1 \ldots u_\ell$ back onto the stack at the same time and end in state r.

CFG to PDA

- We implement this action $(a,s \to u_1 \ldots u_\ell)$ by introducing new states $q_1,\ldots,q_{\ell-1}$ and setting the transition function as follows:

$\delta(q,a,s) \ni (q_1,u_\ell)$,

$\delta(q_1,\varepsilon,\varepsilon) = \{(q_2,u_{\ell-1})\}$,

$\delta(q_2,\varepsilon,\varepsilon) = \{(q_3,u_{\ell-2})\}$,

...

$\delta(q_{\ell-1},\varepsilon,\varepsilon) = \{(r,u_1)\}$.

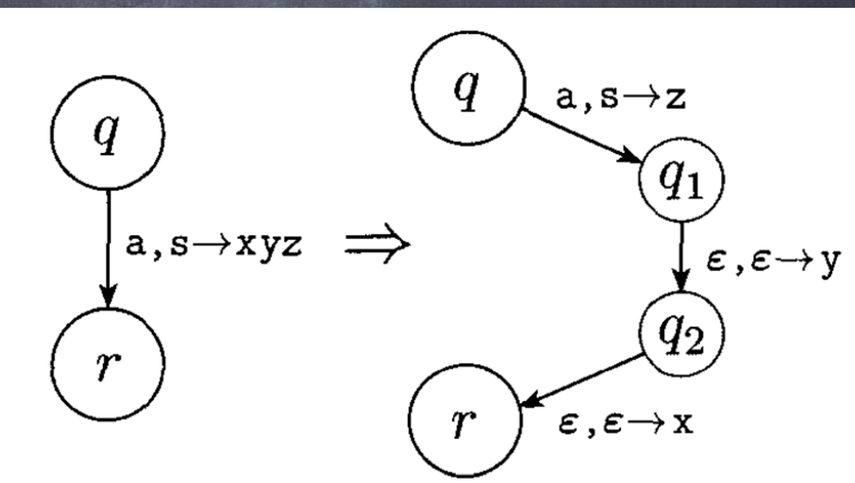
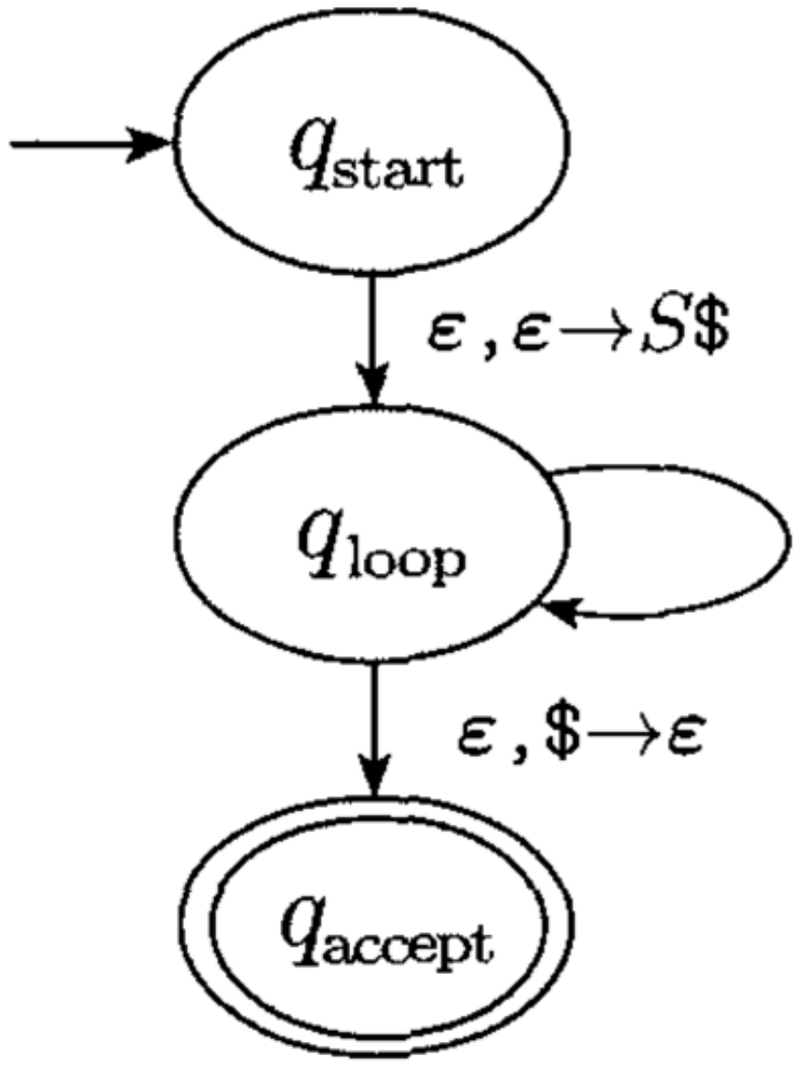


FIGURE 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

CFG to PDA



$\varepsilon, A \to w$ for rule $A \to w$

$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

FIGURE 2.24
State diagram of P

CFG to PDA

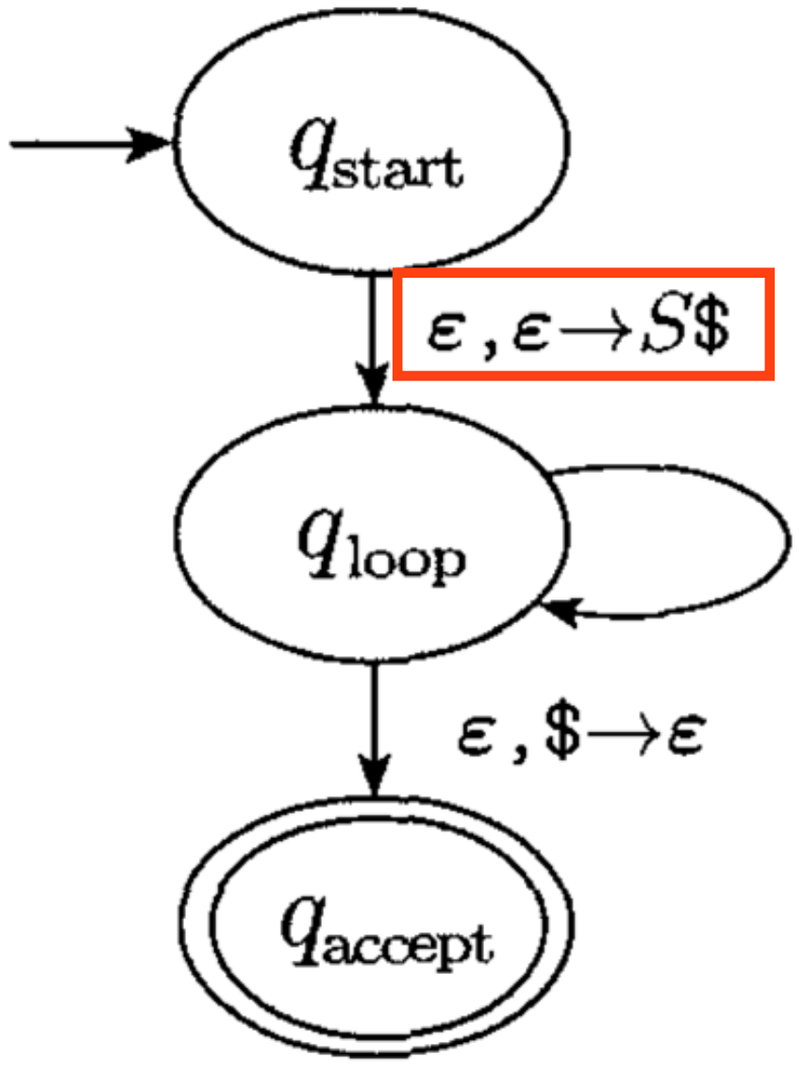
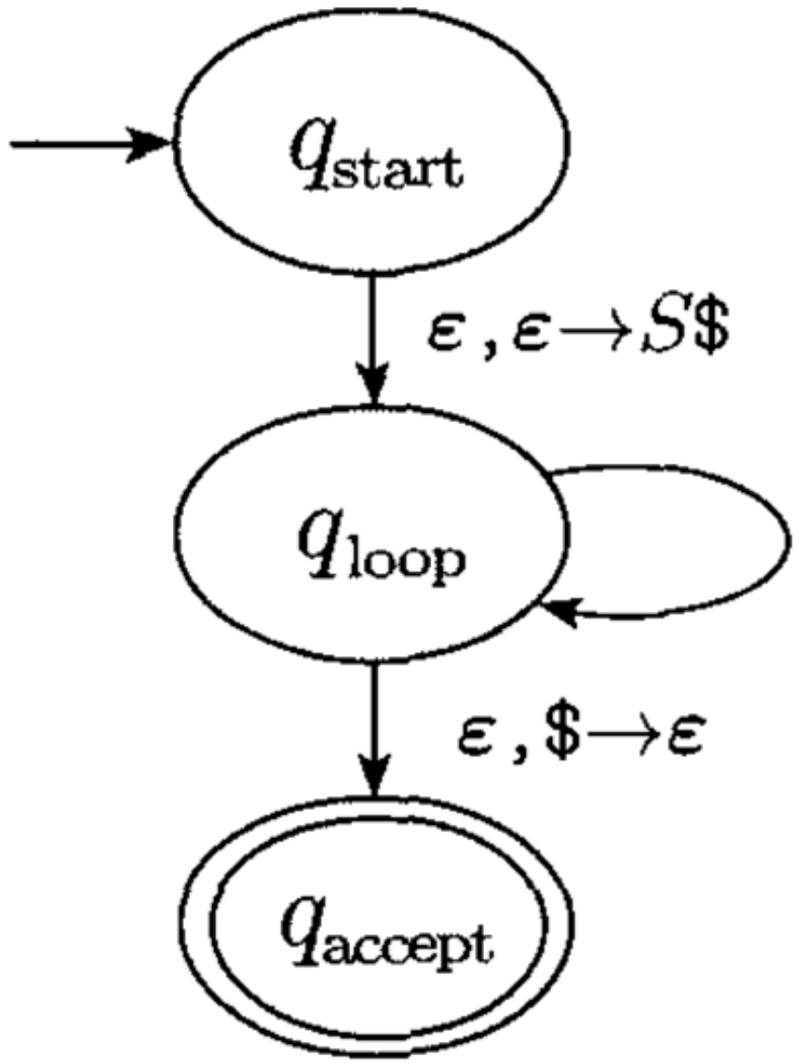


FIGURE 2.24
State diagram of P

CFG to PDA

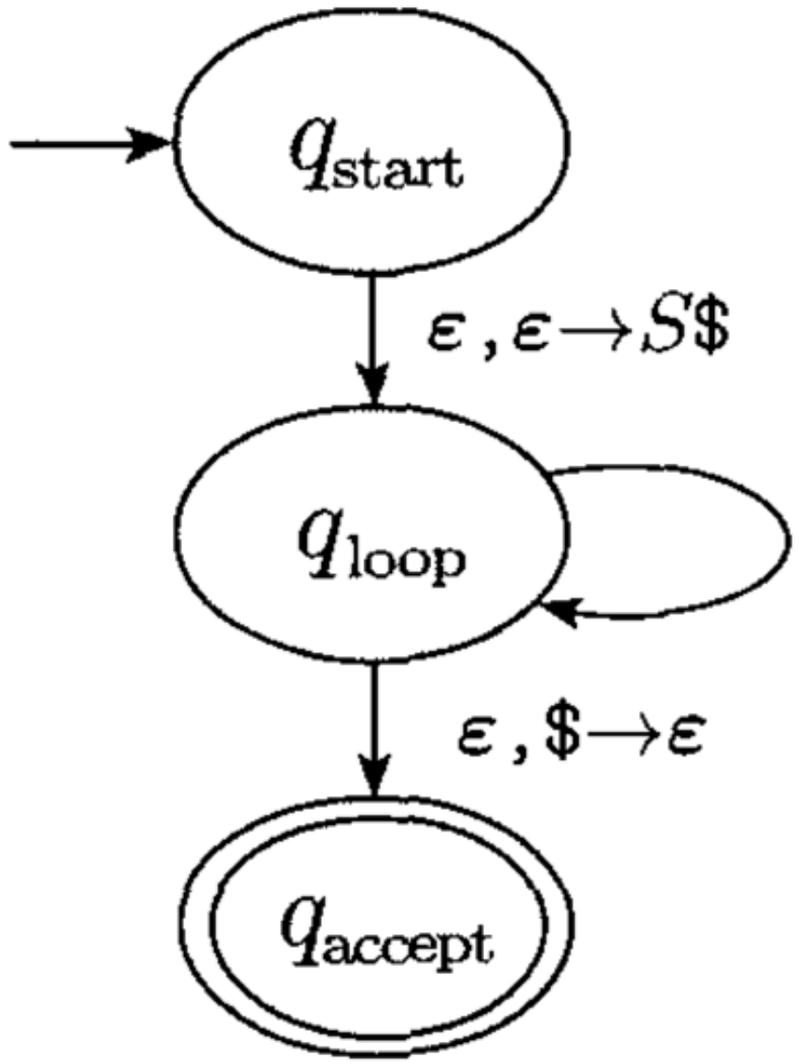


(a)

$\varepsilon, A \to w$	for rule $A \to w$
$\mathtt{a}, \mathtt{a} \to \varepsilon$	for terminal a

FIGURE 2.24
State diagram of P

CFG to PDA



$\varepsilon, A \to w$	for rule $A \to w$
(b)	
$\mathtt{a}, \mathtt{a} \to \varepsilon$	for terminal a

FIGURE 2.24
State diagram of P

CFG to PDA

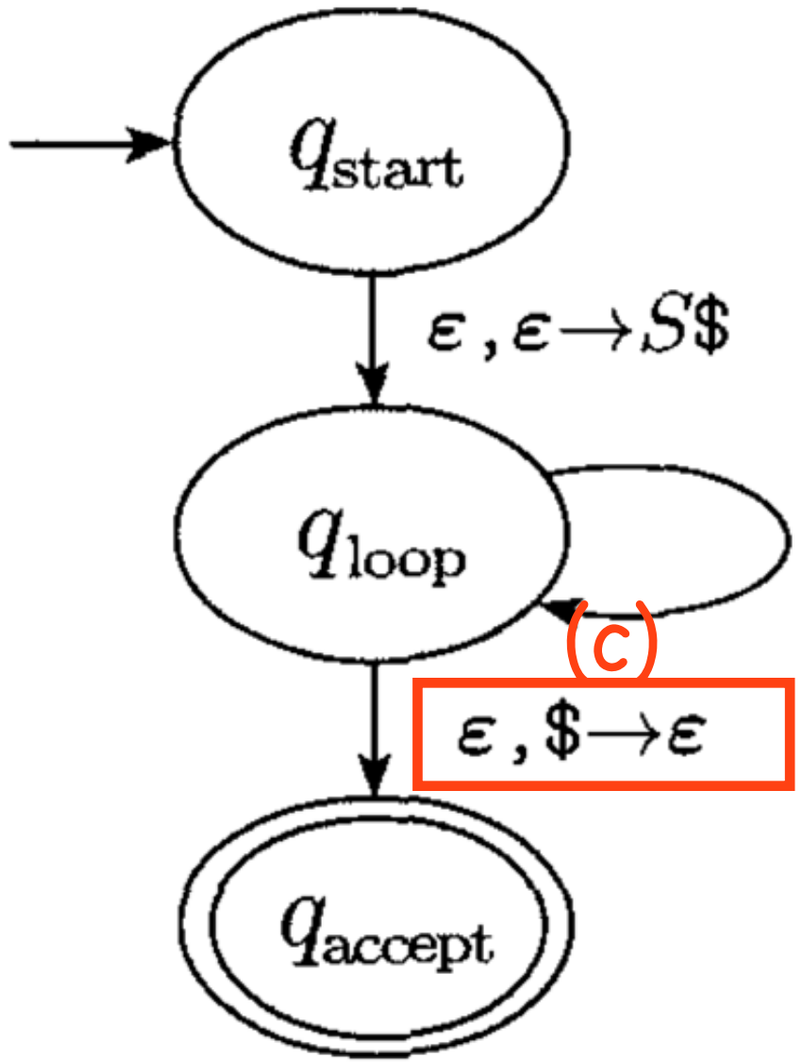
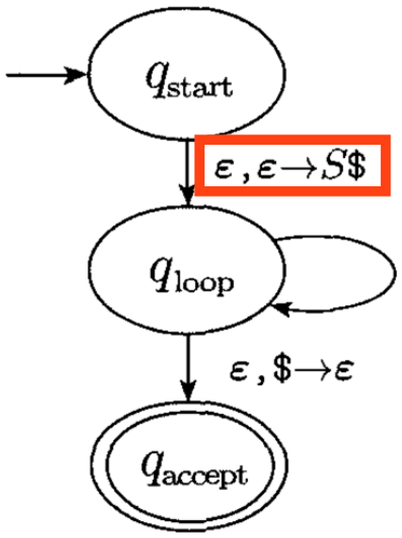


FIGURE 2.24
State diagram of P



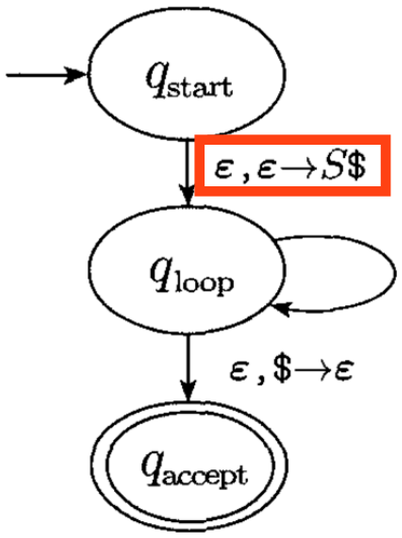
$\varepsilon, A \to w$ for rule $A \to w$

$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

FIGURE 2.24
State diagram of P

CFG to PDA

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.



$\epsilon, A \to w$ for rule $A \to w$

a, a $\to \epsilon$ for terminal a

FIGURE 2.24
State diagram of P

CFG to PDA

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initializing the stack to contain the symbols \$ and S, implementing step 1 in the informal description: $\delta(q_{\text{start}}, \epsilon, \epsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

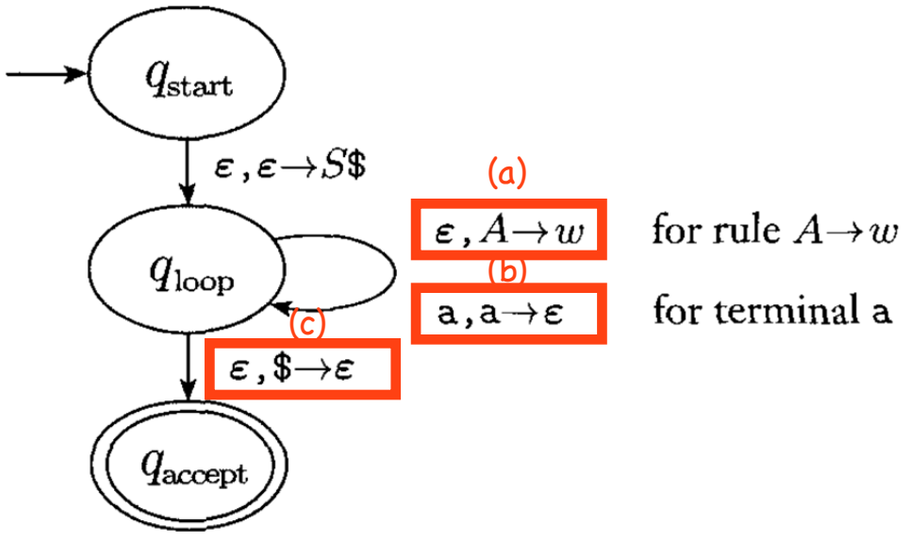


FIGURE 2.24
State diagram of P

CFG to PDA

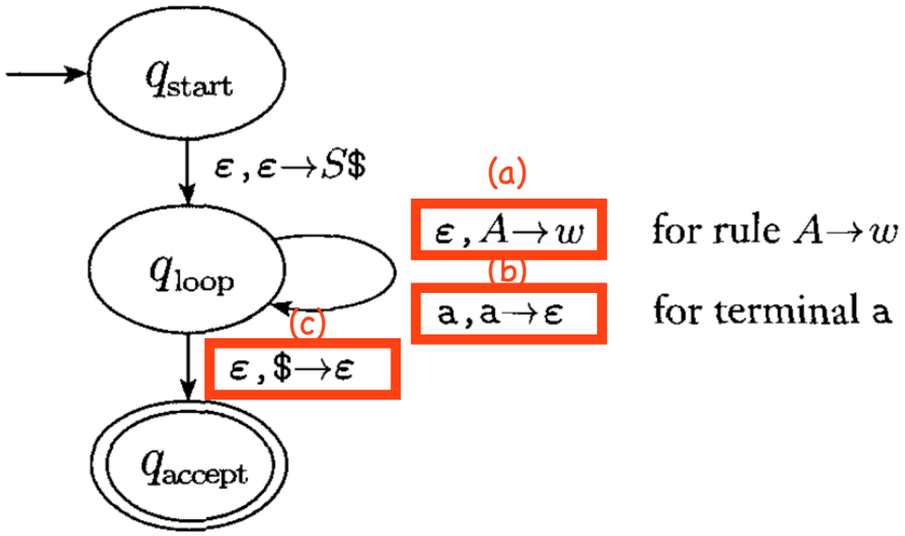


FIGURE 2.24
State diagram of P

CFG to PDA

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}$.

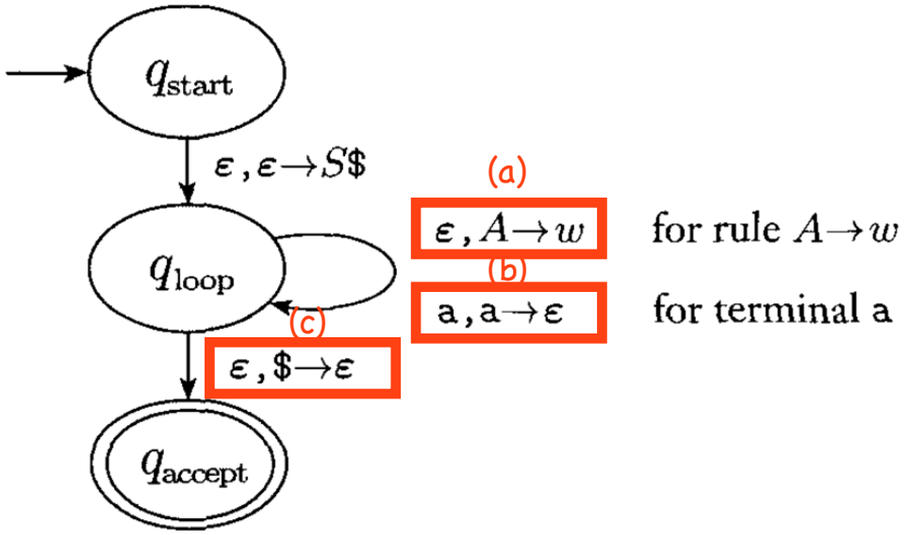


FIGURE 2.24
State diagram of P

CFG to PDA

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

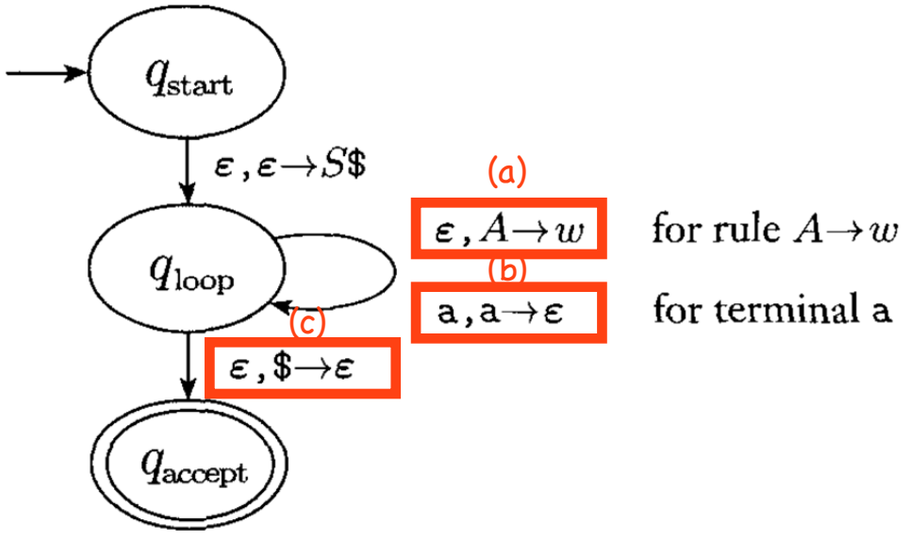


FIGURE 2.24
State diagram of P

CFG to PDA

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker \$ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

CFG to PDA

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

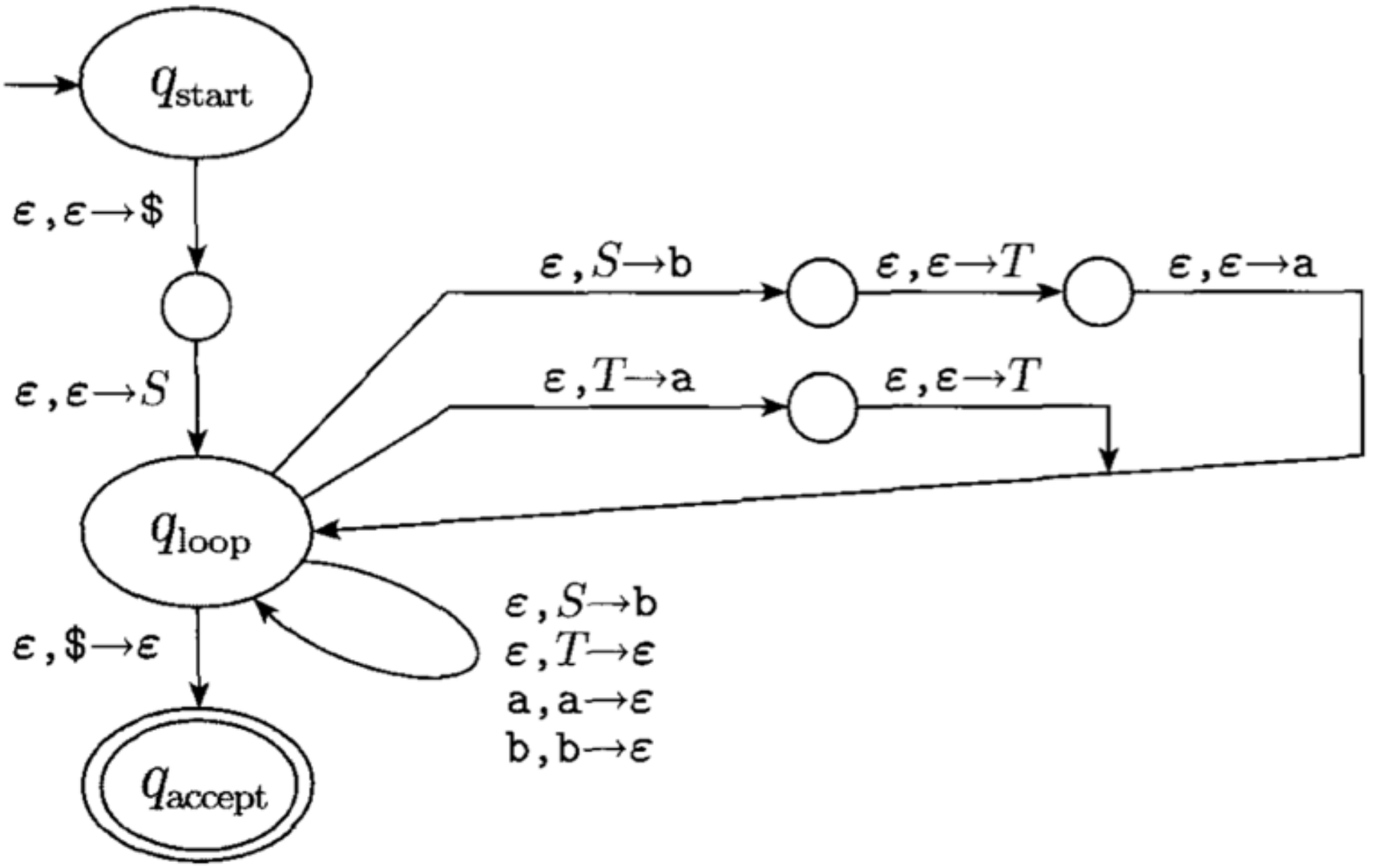


FIGURE 2.26
State diagram of P_1

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

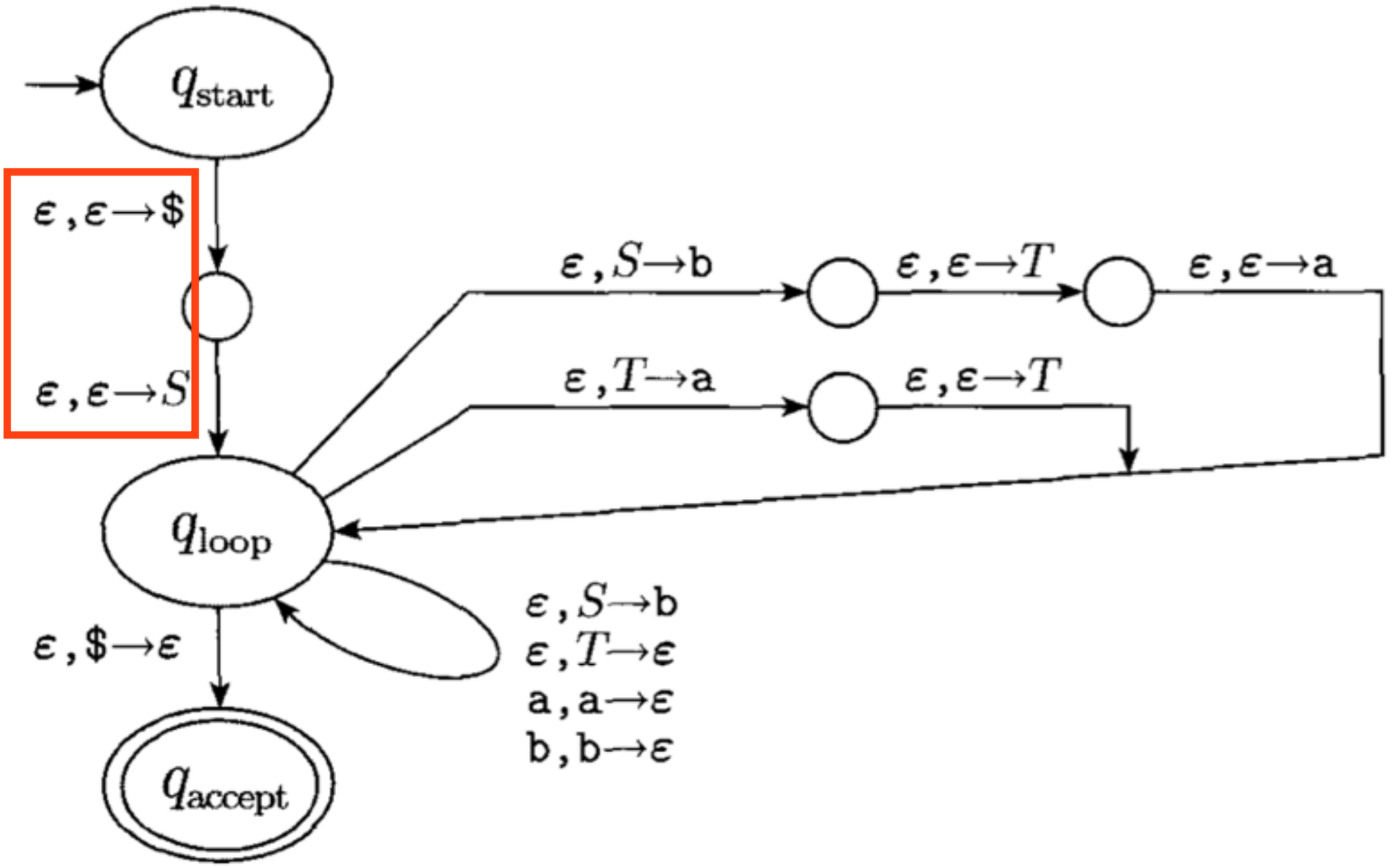


FIGURE 2.26
State diagram of P_1

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

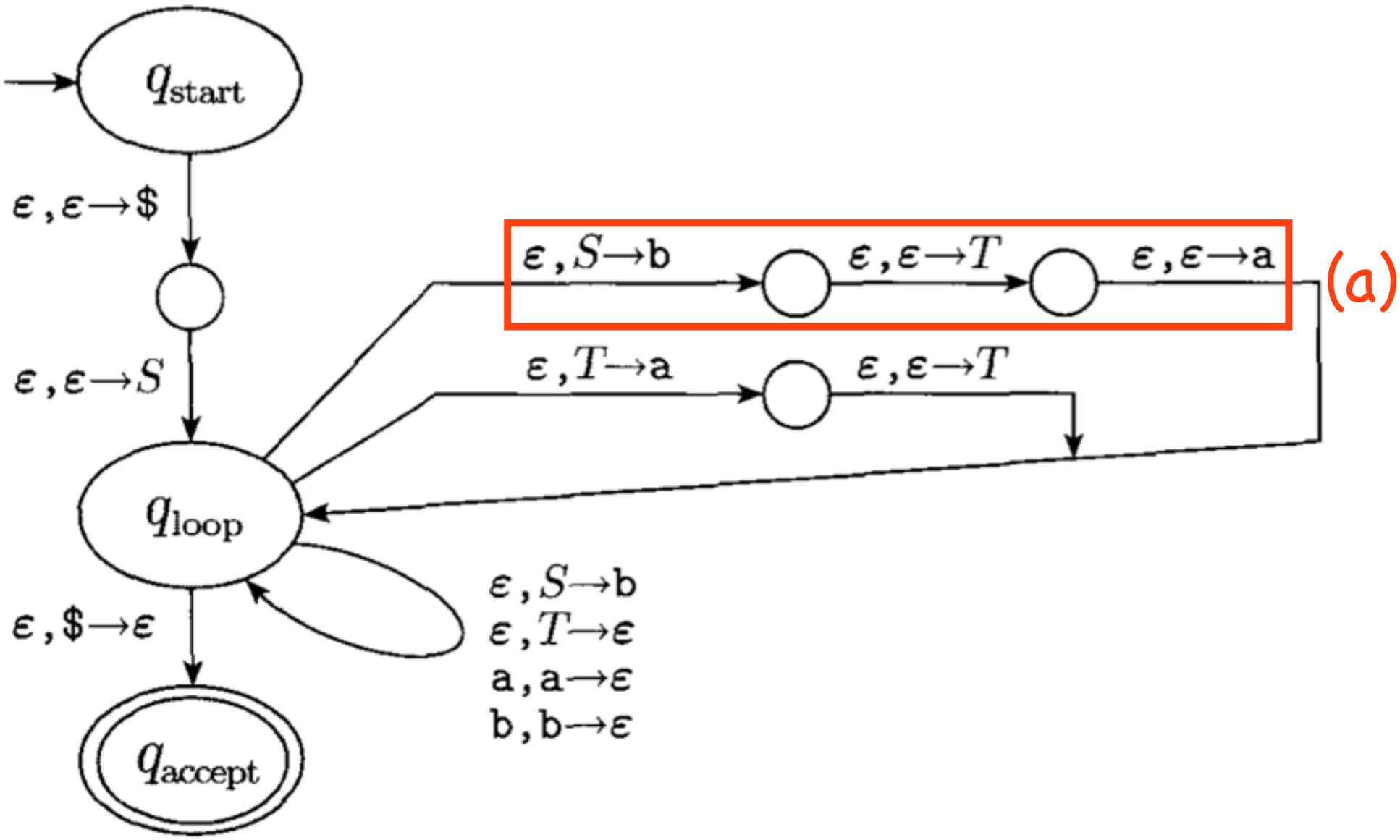


FIGURE 2.26
State diagram of P_1

$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \rightarrow T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

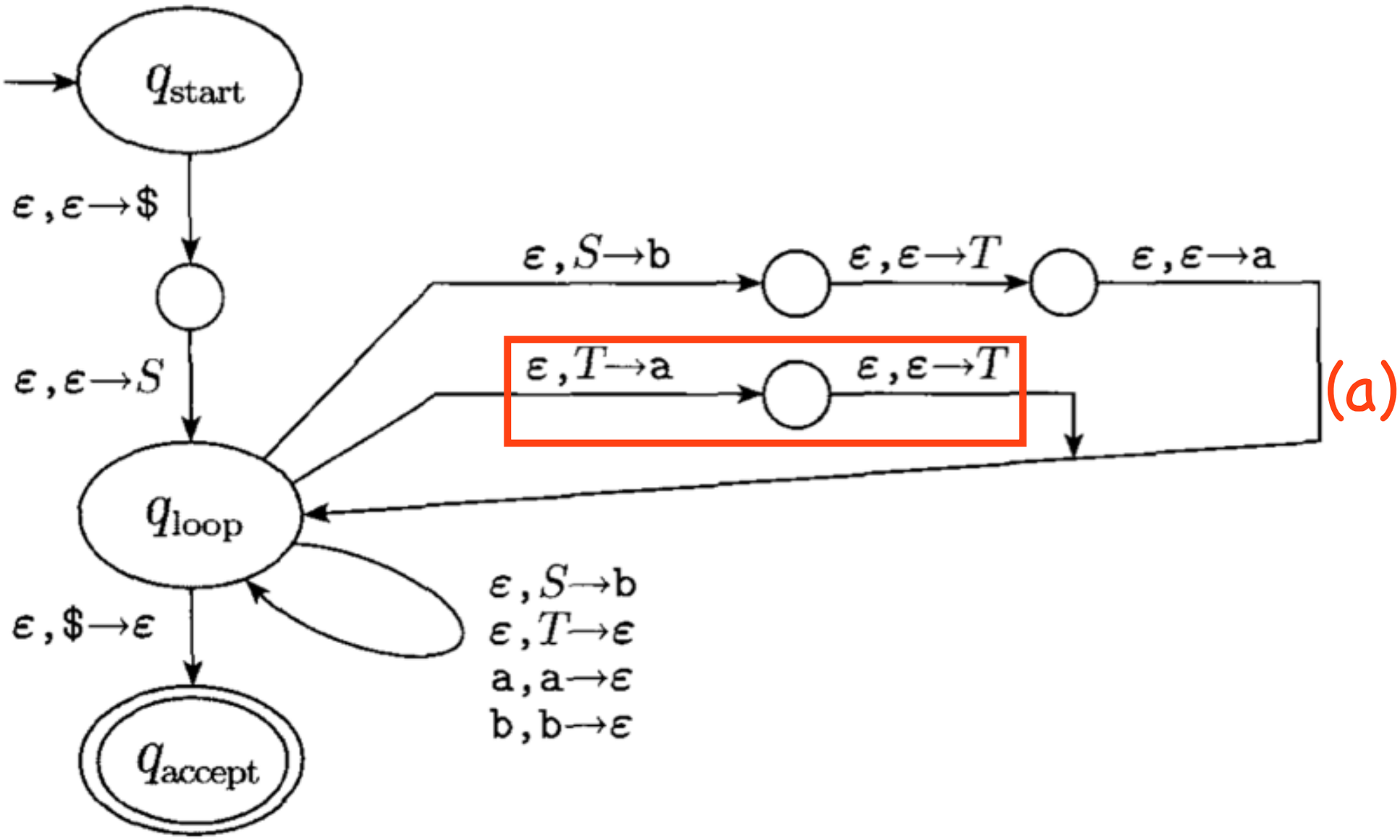


FIGURE 2.26
State diagram of P_1

$$\begin{array}{l} S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b} \\ T \to T\mathtt{a} \mid \varepsilon \end{array}$$

The transition function is shown in the following diagram.

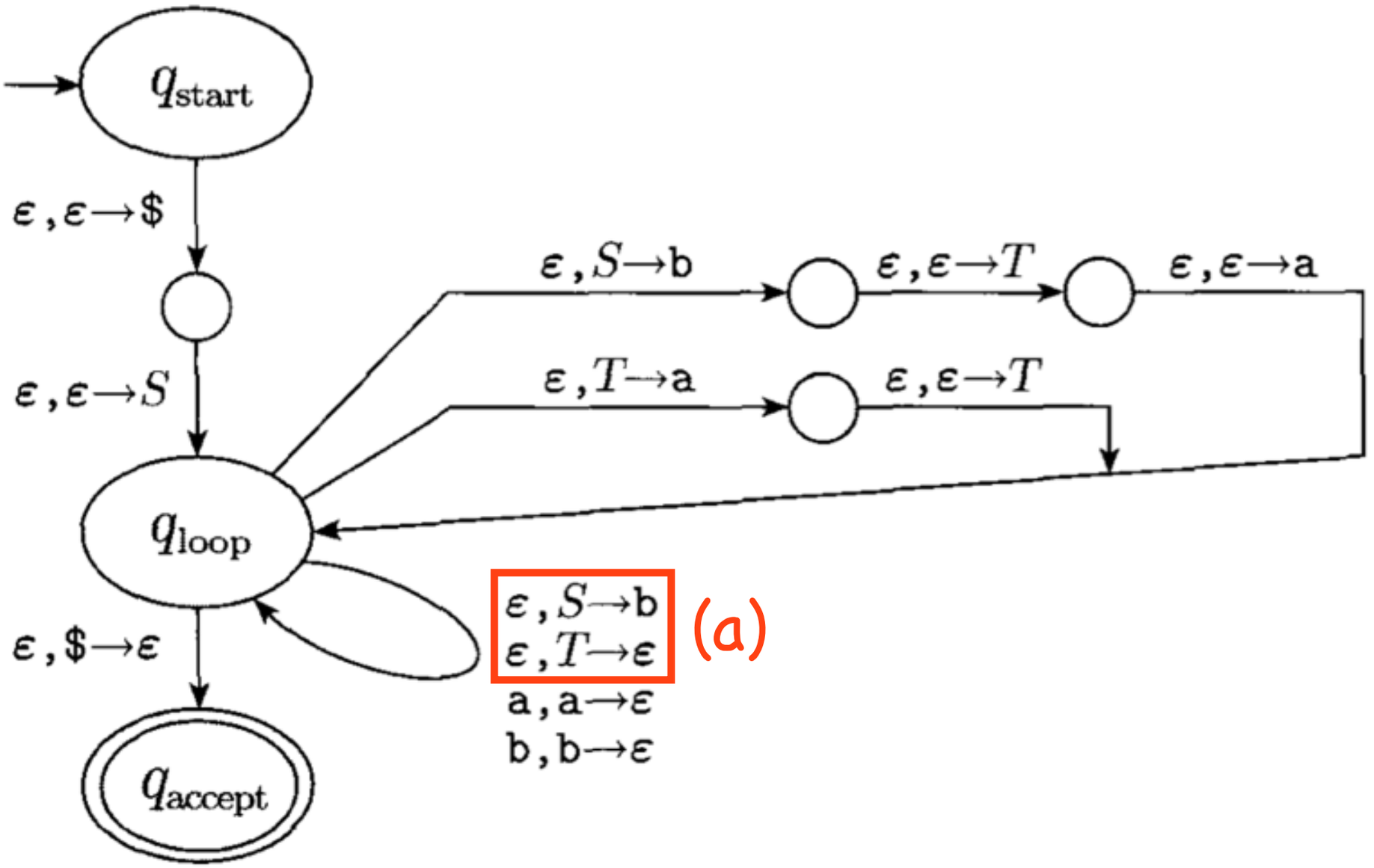


FIGURE 2.26
State diagram of P_1

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

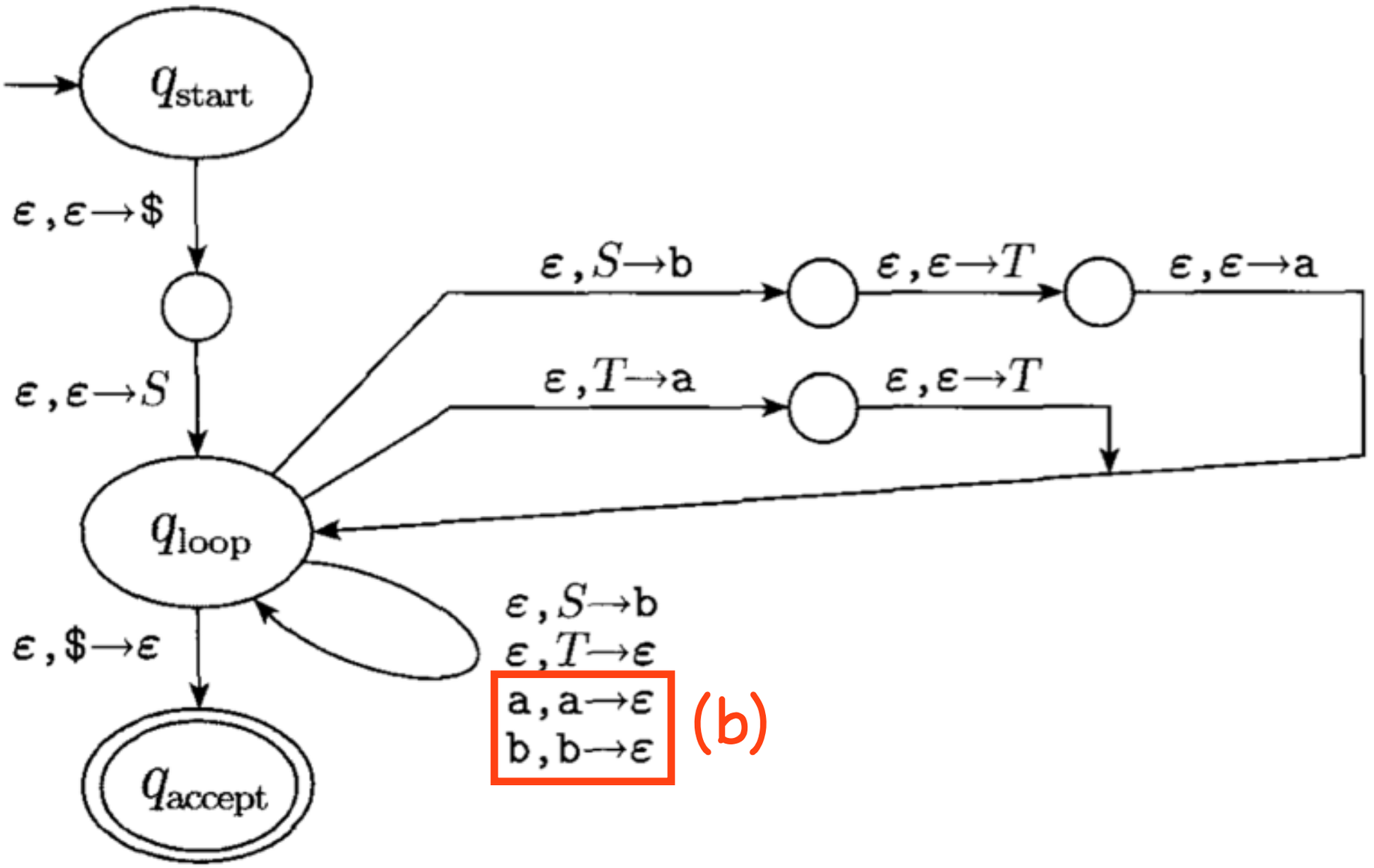


FIGURE 2.26
State diagram of P_1

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.

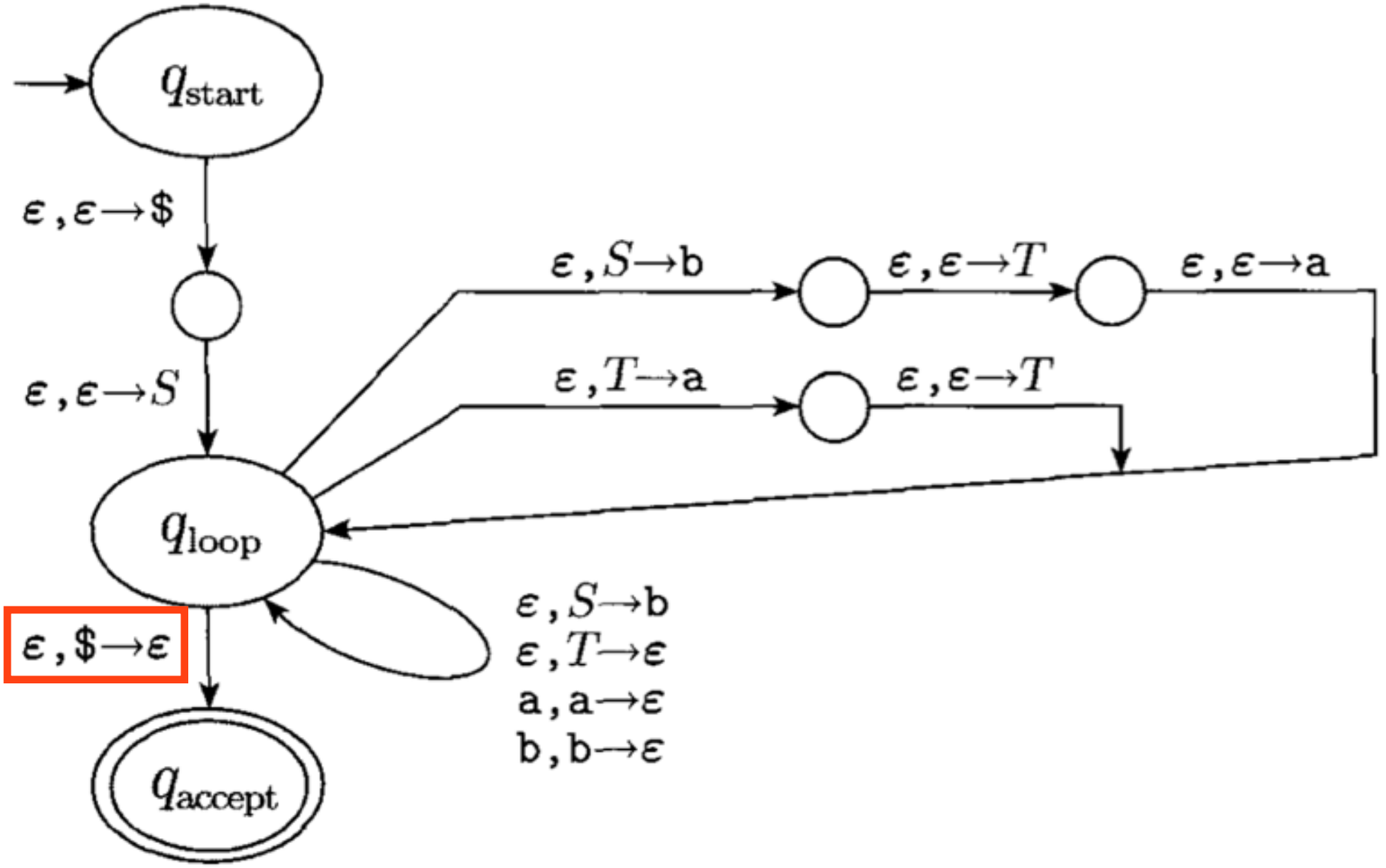


FIGURE 2.26
State diagram of P_1

PDA to CFG

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

PDA to CFG

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

PDA to CFG

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

- Giving P features 1 and 2 is easy.

- To give it feature 3, we replace
each transition that simultaneously pops and pushes with a two-transition sequence that goes through a new state,
each transition that neither pops nor pushes with a two-transition sequence that pushes then pops an arbitrary stack symbol.

PDA to CFG

PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G. The variables of G are $\{A_{pq} | \, p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to a A_{rs} b$ in G.
- For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr} A_{rq}$ in G.
- Finally, for each $p \in Q$, put the rule $A_{pp} \to \varepsilon$ in G.

You may gain some insight for this construction from the following figures.

- For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G.

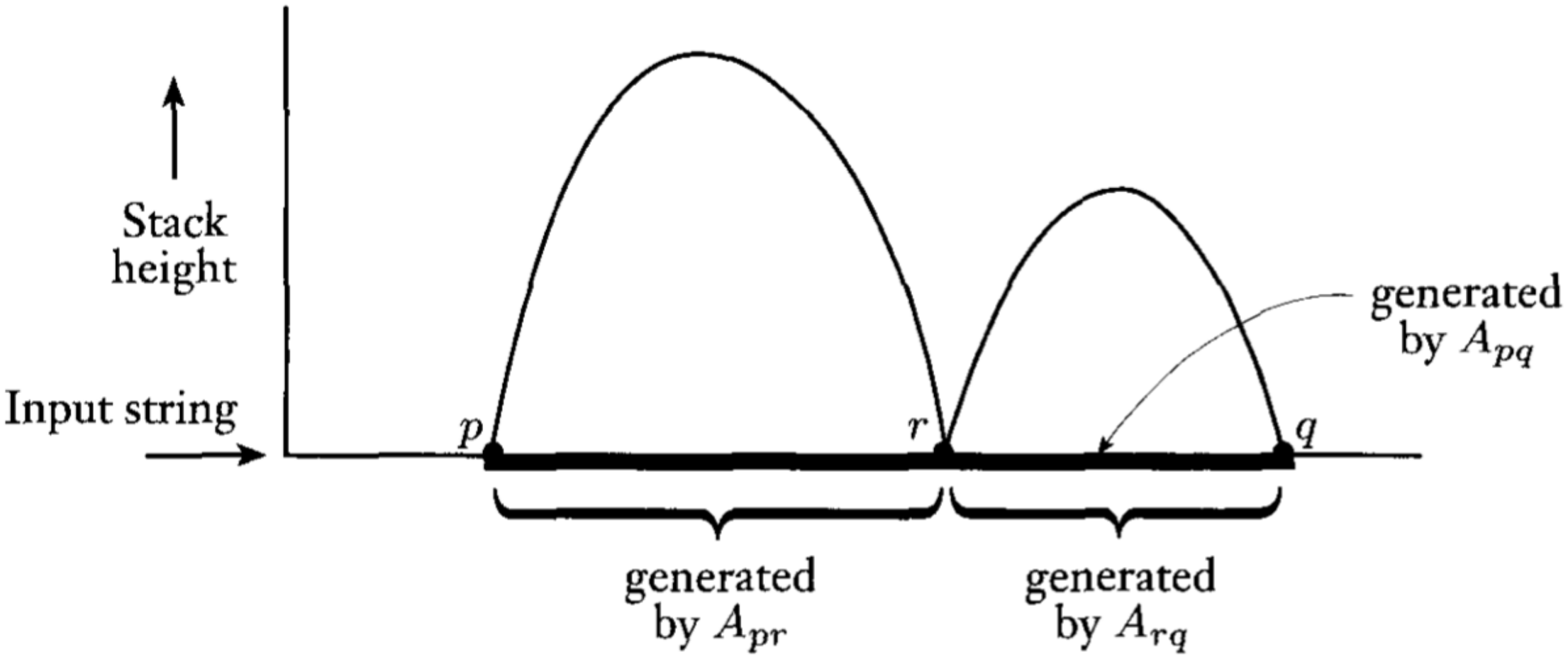


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G.

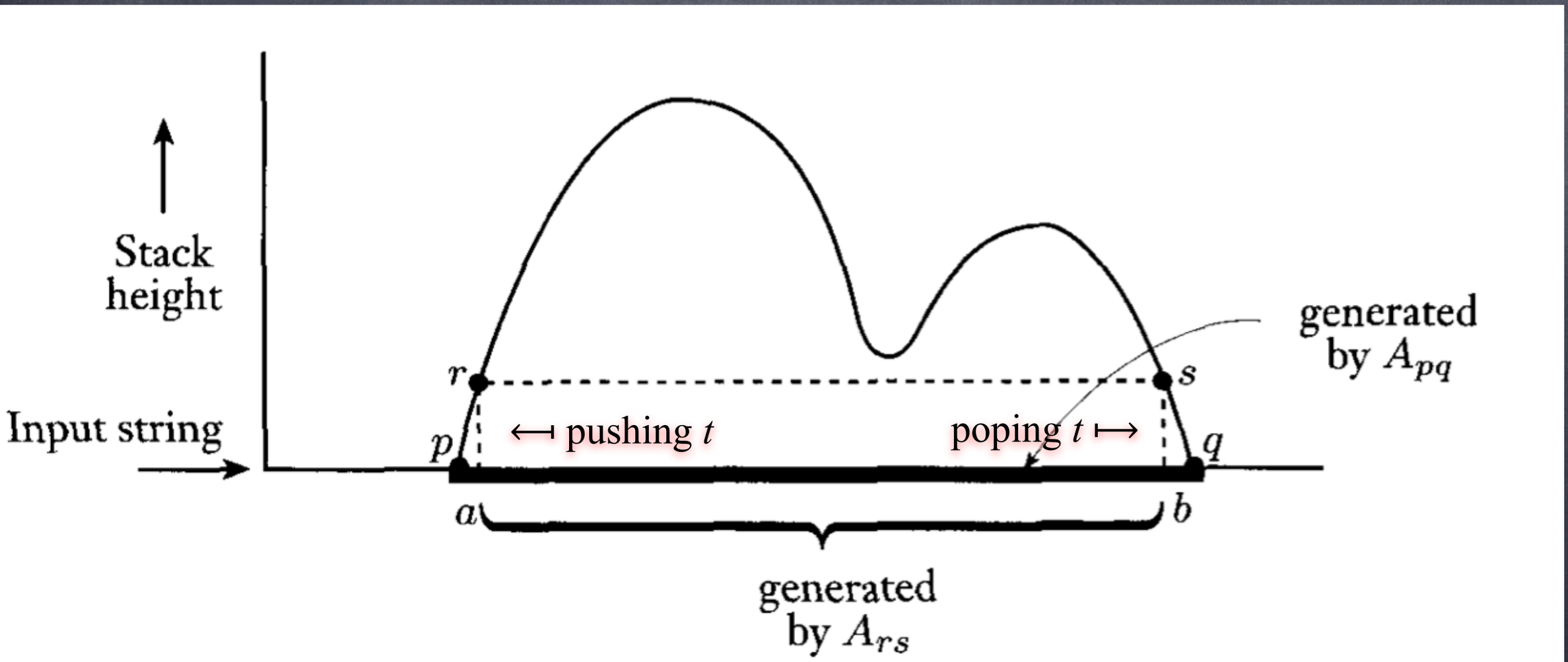


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

PDA to CFG

CLAIM 2.30

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

We prove this claim by induction on the number of steps in the derivation of x from A_{pq}.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

Basis: The derivation has 1 step.

A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \rightarrow \varepsilon$. Clearly, input ε takes P from p with empty stack to p with empty stack so the basis is proved.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

Basis: The derivation has 1 step.
A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \varepsilon$. Clearly, input ε takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k + 1$.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

Basis: The derivation has 1 step.

A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \varepsilon$. Clearly, input ε takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k + 1$.

Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k + 1$ steps. The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. We handle these two cases separately.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \stackrel{*}{\Rightarrow} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack symbol t. Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

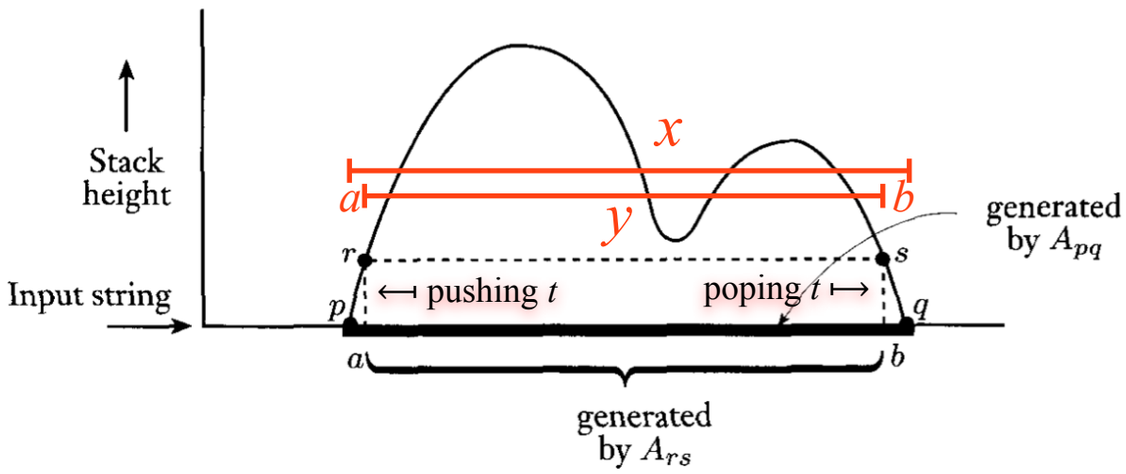


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

$A_{pq} \Rightarrow aA_{rs}b$

In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \overset{*}{\Rightarrow} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack symbol t. Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

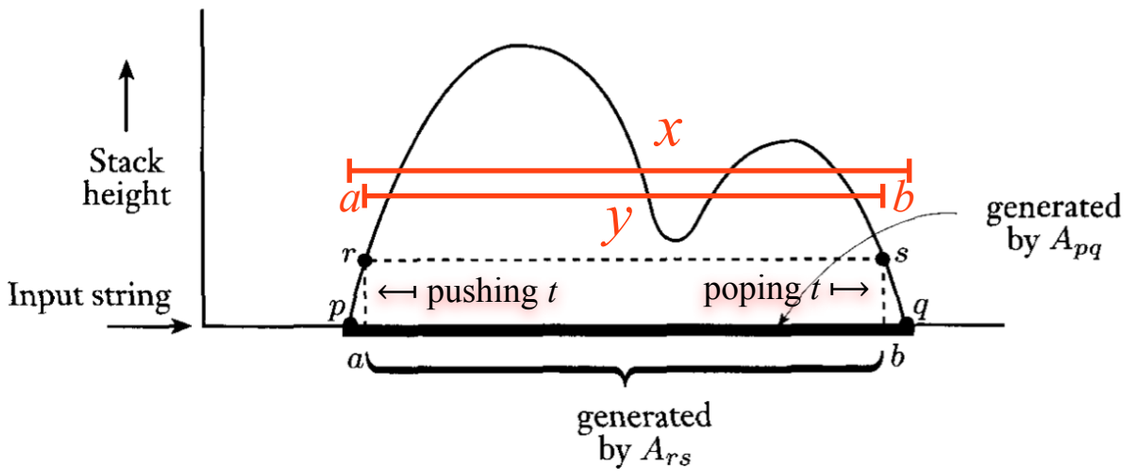


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \overset{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r, and z can bring P from r to q, with empty stacks at the beginning and end. Hence x can bring it from p with empty stack to q with empty stack. This completes the induction step.

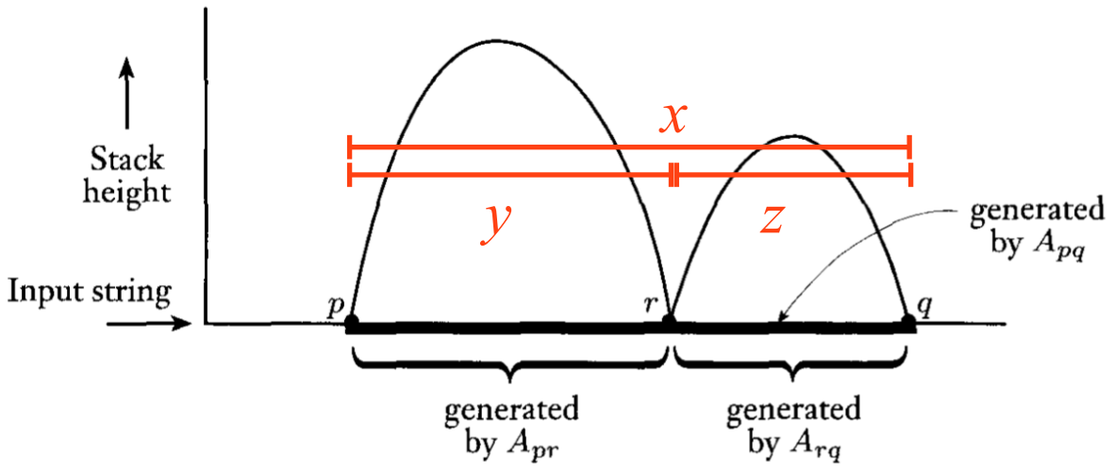


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

$A_{pq} \Rightarrow A_{pr}A_{rq}$

In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \overset{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r, and z can bring P from r to q, with empty stacks at the beginning and end. Hence x can bring it from p with empty stack to q with empty stack. This completes the induction step.

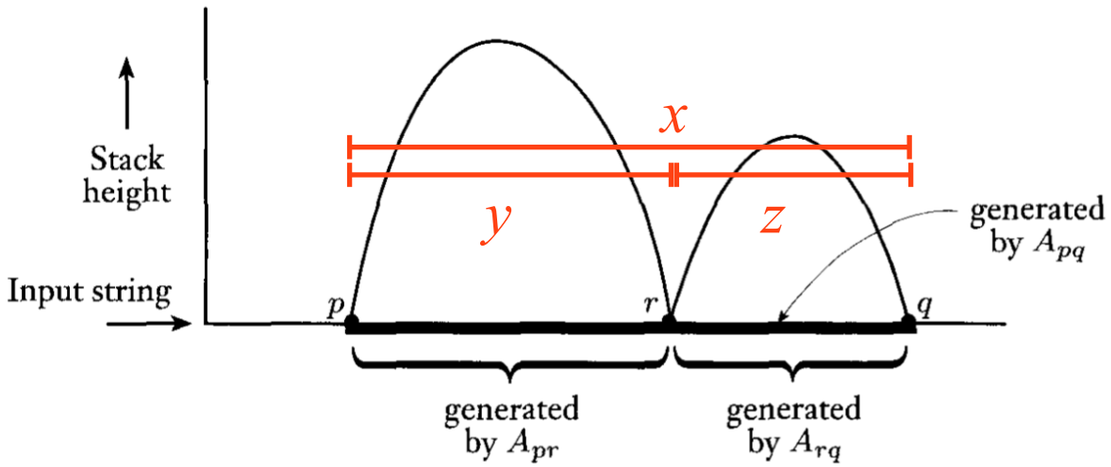


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA to CFG

CLAIM 2.31

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

We prove this claim by induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

Basis: The computation has 0 steps.

If a computation has 0 steps, it starts and ends at the same state—say, p. So we must show that $A_{pp} \stackrel{*}{\Rightarrow} x$. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \rightarrow \epsilon$, so the basis is proved.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

Basis: The computation has 0 steps.

If a computation has 0 steps, it starts and ends at the same state—say, p. So we must show that $A_{pp} \stackrel{*}{\Rightarrow} x$. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \to \epsilon$, so the basis is proved.

Induction step: Assume true for computations of length at most k, where $k \geq 0$, and prove true for computations of length $k + 1$.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

Basis: The computation has 0 steps.

If a computation has 0 steps, it starts and ends at the same state—say, p. So we must show that $A_{pp} \stackrel{*}{\Rightarrow} x$. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \to \epsilon$, so the basis is proved.

Induction step: Assume true for computations of length at most k, where $k \geq 0$, and prove true for computations of length $k + 1$.

Suppose that P has a computation wherein x brings p to q with empty stacks in $k + 1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

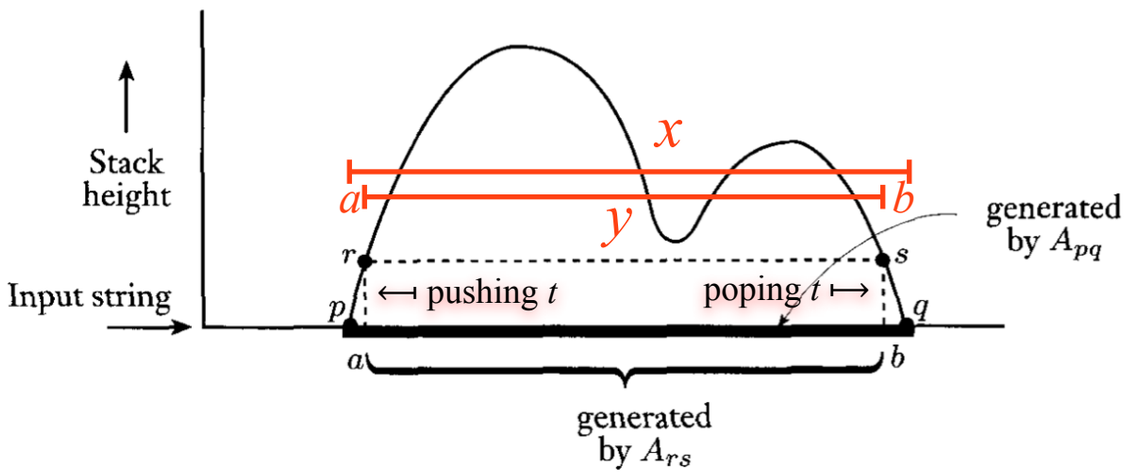


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G.

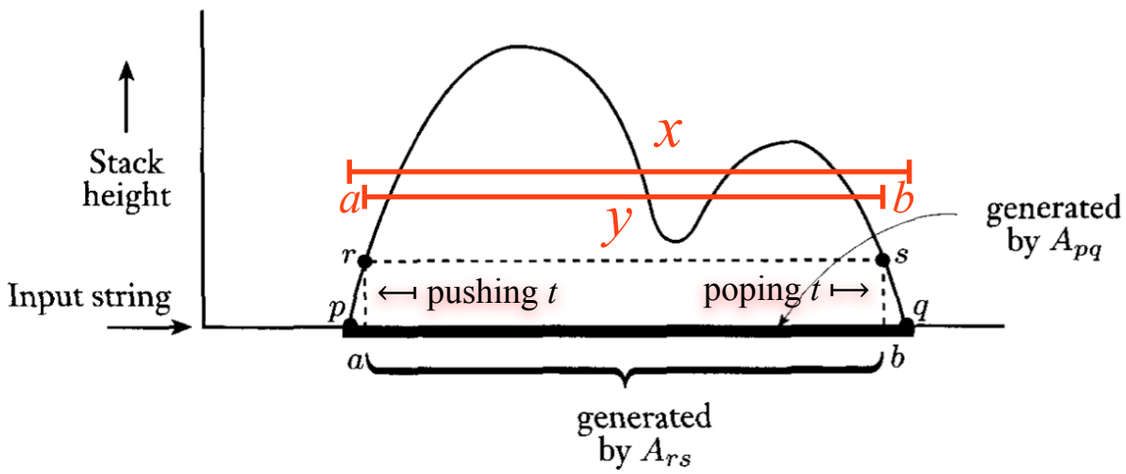


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

the stack is empty only at the beginning and end

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G.

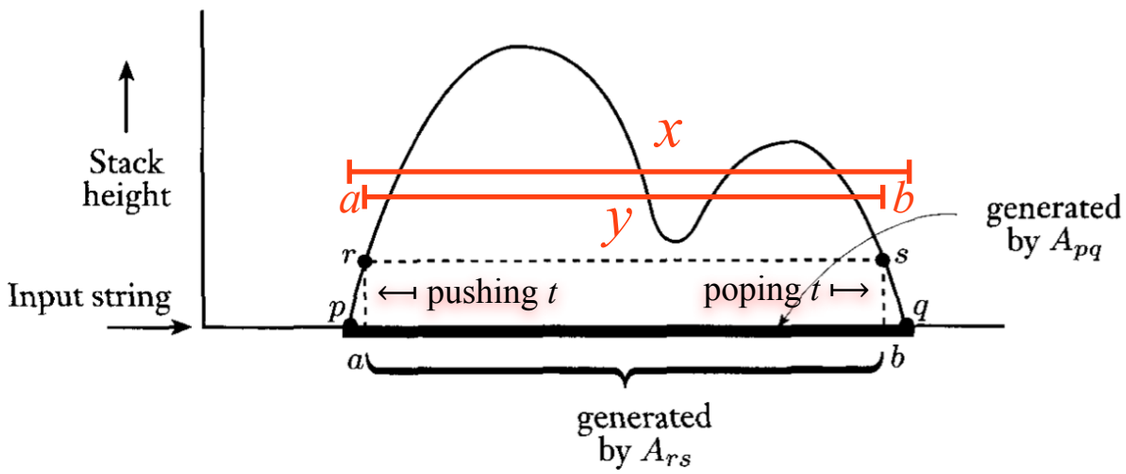


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

the stack is empty only at the beginning and end

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G.

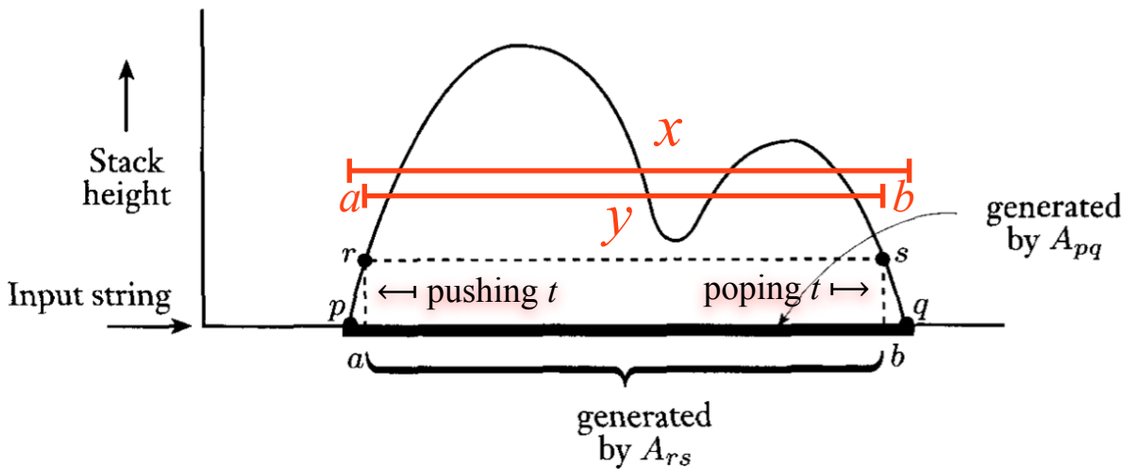


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

the stack is empty only at the beginning and end

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G.

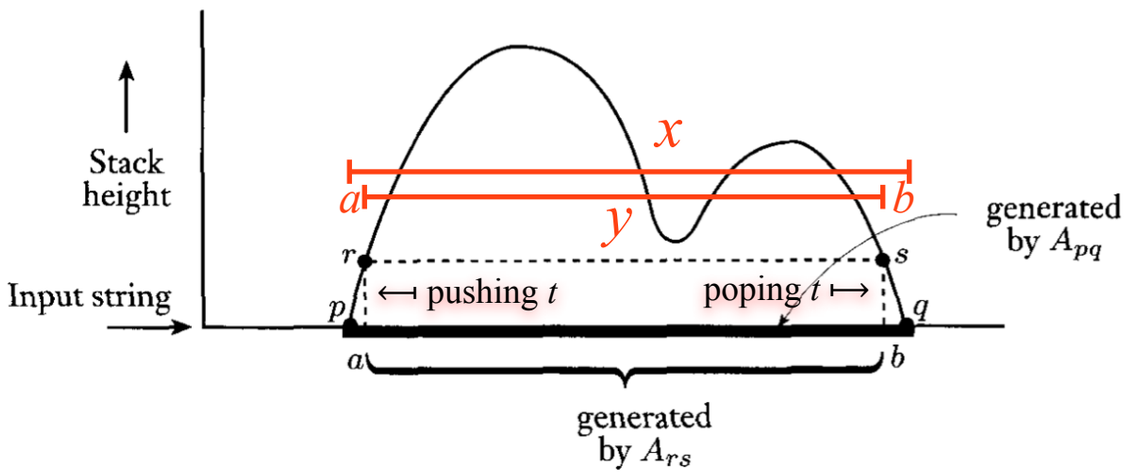


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

the stack is empty only at the beginning and end

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \to aA_{rs}b$ is in G.

Let y be the portion of x without a and b, so $x = ayb$. Input y can bring P from r to s without touching the symbol t that is on the stack and so P can go from r with an empty stack to s with an empty stack on input y. We have removed the first and last steps of the $k+1$ steps in the original computation on x so the computation on y has $(k+1) - 2 = k - 1$ steps. Thus the induction hypothesis tells us that $A_{rs} \overset{*}{\Rightarrow} y$. Hence $A_{pq} \overset{*}{\Rightarrow} x$.

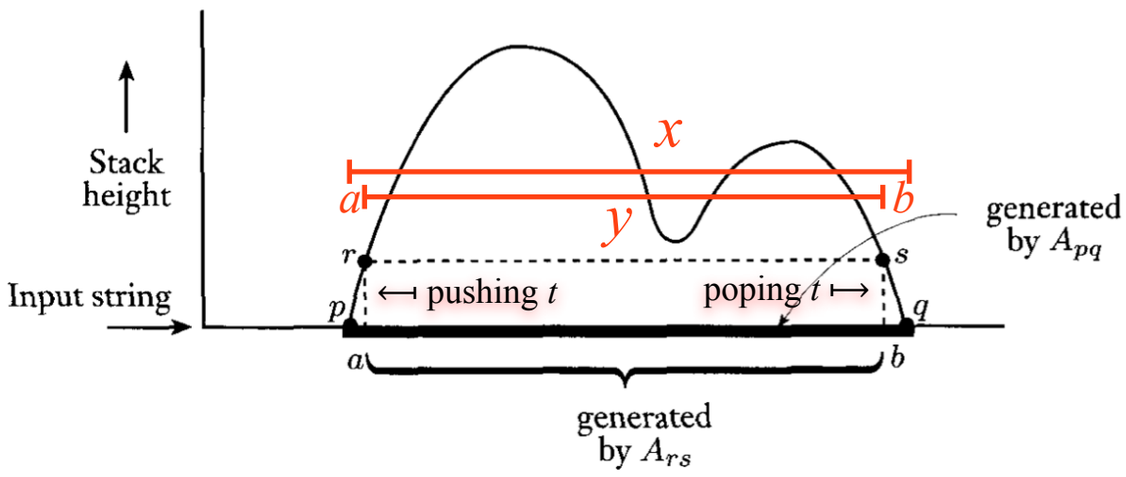


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \to aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x. Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input read during the first portion and z is the input read during the second portion. The induction hypothesis tells us that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G, $A_{pq} \overset{*}{\Rightarrow} x$, and the proof is complete.

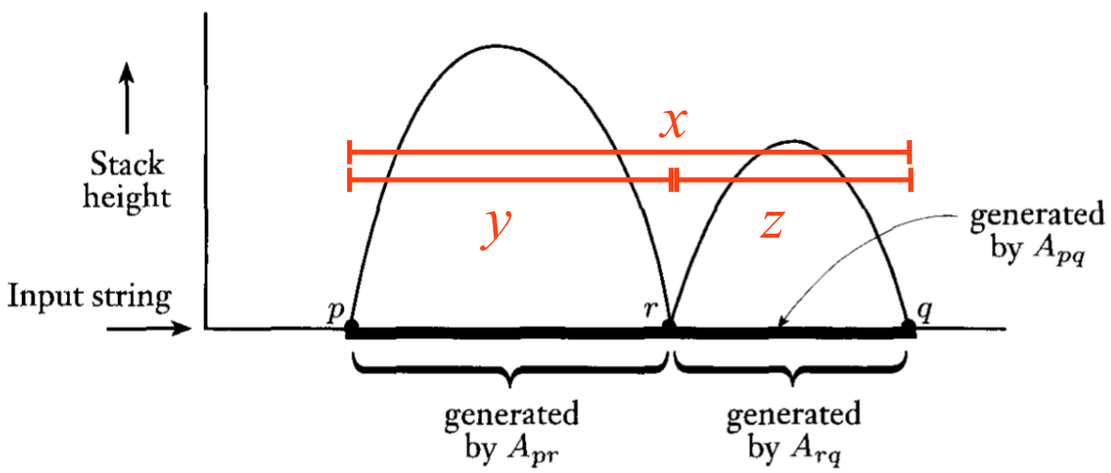


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

it becomes empty elsewhere, too.

In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x. Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input read during the first portion and z is the input read during the second portion. The induction hypothesis tells us that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G, $A_{pq} \overset{*}{\Rightarrow} x$, and the proof is complete.

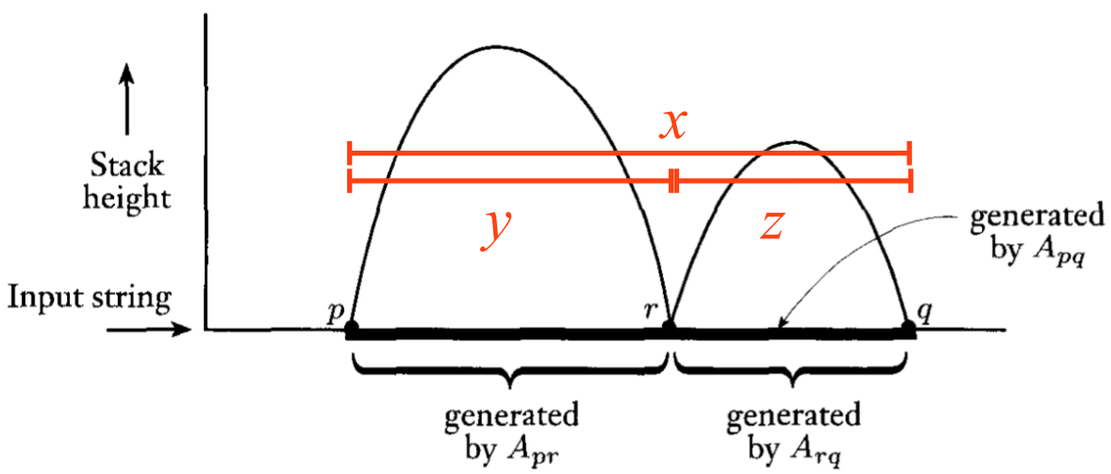


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA vs CFG

PDA vs CFG

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

PDA vs CFG

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

PDA vs CFG

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

COMP-330
Theory of Computation

Fall 2019 -- Prof. Claude Crépeau

Lec. 12 :
PDA-CFG equivalence