Context-Free Languages

Mohamed Kobeissi

October 26, 2018

Grammars

A grammar is a 4-tuple (V, Σ, R, S), where

- V is a finite set called the **variables**
- Σ is a finite set, disjoint form V, called the **terminals**
- R is a set of **rules**, with each rule being a variable and a string of variables and terminals, and
- $S \in V$ is the start variable

- If u, v, and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv **yields** uwv
- We say that u **derives** v , written $u \stackrel{*}{\Rightarrow} v$ if $u = v$, or if a sequence $u_1, u_2, \ldots, u_k$, exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

Such a sequence is called a **derivation** or **parse**, and discovering the derivation is called **parsing**

Language of a grammar

- The language generated by a grammar is the set of all strings of terminals produced from S using rules (or productions) as substitutions
- The language of a grammar is defined by:

$$\{w \in \Sigma^*; S \stackrel{*}{\Rightarrow} w\}$$

Parse Trees

Parse trees are derivations of strings from grammars. The root node in the tree is the start symbol, every internal node is a nonterminal, and leaf nodes are read left to right to give the string corresponding to the tree. Parse trees show the structure of an expression as it corresponds to the grammar

Regular Grammars

A grammar is **regular** if all the rules have one of the following forms:

$$A \to aB$$

$$A \to a$$

$$A \to \lambda$$

where A, B are in V and $a \in \Sigma$

Example:
$G = (\{S\}, \{x, y, z\}, R, S)$, where the productions of R are:

$$S \to xS$$
$$S \to y$$
$$S \to z$$

The above grammar can be equivalently defined by:

$$S \to xS|y|z$$

Example:

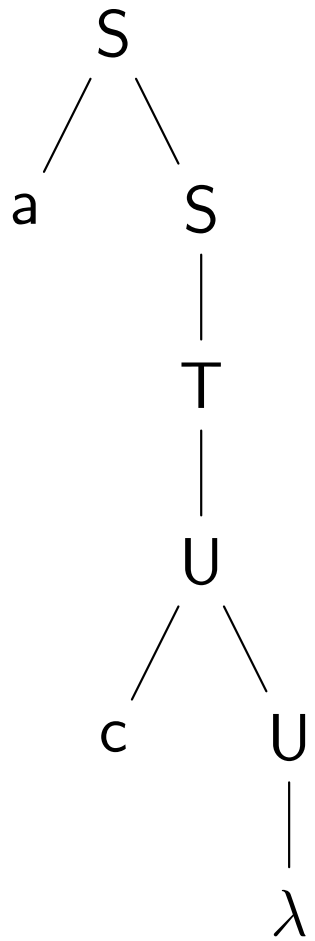
For the grammar

$$S \to aS|T$$
$$T \to bT|U$$
$$U \to cU|\lambda$$

we have the derivation

$S \implies aS \implies aT \implies aU \implies acU \implies ac$

The parse tree for the derivation is:



Proposition: A language is regular iff it is generated by a regular grammar

Example:

Find the language defined by the grammar G below:

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow \lambda \end{aligned}$$

Note that the grammar can be equivalently defined by:

$$S \rightarrow aS|\lambda$$

Example:

The grammar

$$S \rightarrow aS|bS|\lambda$$

generates all the words in $\{a, b\}^*$

Grammars for regular languages

Constructing a grammar for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent regular grammar as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the grammar if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \lambda$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine

Find a regular grammar for the language

$L = \{w \in \{a, b\}^* \ ; \ \text{every } a \text{ in } w \text{ is followed by at least one } b\}$

Example:

- Find the language defined by the grammar G

$$\begin{aligned} S &\rightarrow XaaX \\ X &\rightarrow aX|bX|\lambda \end{aligned}$$

  Then give a derivation of the string $ababaaaba$
- Is the grammar G regular ? if not give a regular grammar whose language is $L(G)$

Context-Free Grammars

- A grammar G is context free is the rules are in $(V \cup \Sigma)^*$
- A language generated by a CFG is a context-free language (CFL)

CFG for non-regular languages

Example:

The CFG G:

$$S \rightarrow aSb|\lambda$$

generates the language $L = \{a^n b^n; n \geq 0\}$

Proof: $L(G) \subseteq L$?

$L \subseteq L(G)$?

Designing Context-Free Grammars

The grammar

$$S_1 \rightarrow 0S_1 1|\lambda$$

generates the language $\{0^n 1^n; n \geq 0\}$

and the grammar

$$S_2 \rightarrow 1S_2 0|\lambda$$

generates the language $\{1^n 0^n; n \geq 0\}$

CFG for union

The CFG

$$\begin{aligned} S &\rightarrow S_1|S_2 \\ S_1 &\rightarrow 0S_1 1|\lambda \\ S_2 &\rightarrow 1S_2 0|\lambda \end{aligned}$$

generates the language $\{0^n 1^n; n \geq 0\} \cup \{1^n 0^n; n \geq 0\}$

CFG for concatenation

The CFG

$$\begin{aligned} S &\rightarrow S_1 S_2 \\ S_1 &\rightarrow 0S_1 1 | \lambda \\ S_2 &\rightarrow 1S_2 0 | \lambda \end{aligned}$$

generates the language $\{0^n 1^n 1^m 0^m; m, n \geq 0\}$

CFG for Kleene star

Give CFG that generates the language L^*, where

$$L = \{0^n 1^n; n \geq 0\}$$

Example:

Show that the CFG

$$S \rightarrow aSb|SS|\lambda$$

generates all nested brackets
example: $aabb = (())$

Example:

Let G be the grammar

$$S \to aSb|bSa|SS|\lambda$$

and

$$L = \{w \in \{a, b\}^*; |w|_a = |w|_b\}$$

Show that $L(G) = L$

Example:

Give CFG that accepts all the words on $\{0,1\}^*$ that starts and ends with the same symbol

Leftmost derivation

A derivation of a string w in a grammar G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced

Example: For the CFG

$$\begin{aligned} S &\rightarrow XaaX \\ X &\rightarrow aX|bX|\lambda \end{aligned}$$

The derivation

$$S \Rightarrow XaaX \Rightarrow aXaaX \Rightarrow aaaX \Rightarrow aaabX \Rightarrow aaab$$

is leftmost

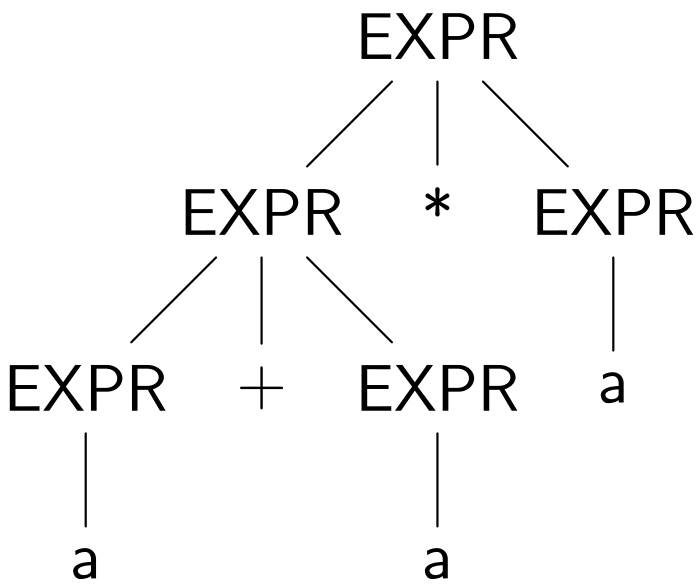
Ambiguity

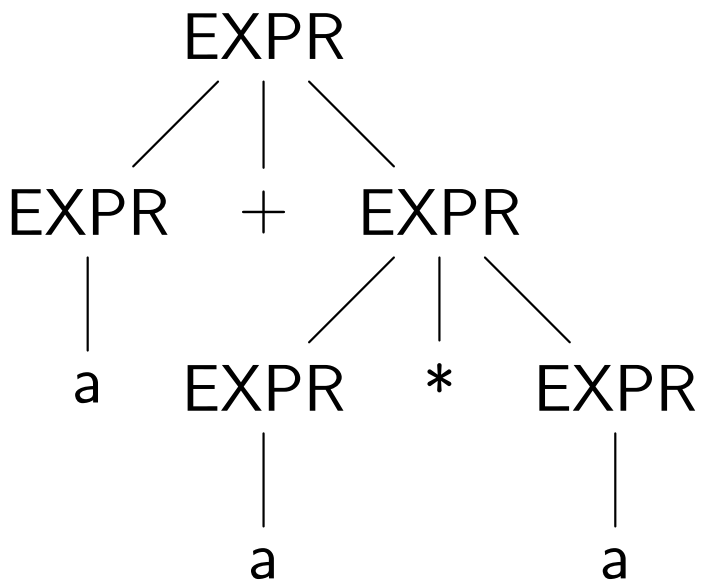
If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar. If a grammar generates some string ambiguously, we say that the grammar is ambiguous

Example: The CFG

$$< EXPR > \rightarrow < EXPR > + < EXPR >$$
$$< EXPR > \rightarrow < EXPR > \times < EXPR > | (< EXPR >) | a$$

is ambiguous





are two parse trees for the string $a + a \times a$

A string w is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously

Example: The CFG

$$A \rightarrow aA|Aa|\lambda$$

is ambiguous. Two (leftmost) derivations of the strings aa are:

$$A \Rightarrow aA \Rightarrow aaA \Rightarrow aa$$

and

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow aa$$

Chomsky Normal Form

A context-free grammar is in **Chomsky normal form** if every rule is of the form

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

where a is any terminal and A, B, and C are any variables, except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \lambda$, where S is the start variable

Some rules of grammar in Chomsky's normal form are:

- The derivation of a string of length n requires $2n-1$ steps
- The parse tree of a derivation is a binary tree whose height (depth) is at most the length of the string

Proof: (from Sipser)

First, we add a new start variable S_0 and the rule $S_0 \rightarrow S$, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule

Second, we take care of all λ-rules. We remove a λ-rule $A \rightarrow \lambda$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \rightarrow uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \rightarrow uv$

We do so for each occurrence of an A, so the rule $R \to uAvAw$ causes us to add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$. If we have the rule $R \to A$, we add $R \to \lambda$ unless we had previously removed the rule $R \to \lambda$. We repeat these steps until we eliminate all λ-rules not involving the start variable

Third, we handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules

Finally, we convert all remaining rules into the proper form. We replace each rule $A \to u_1 u_2 \ldots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \to u_1 A_1$, $A_1 \to u_2 A_2$, $A_2 \to u_3 A_3, \ldots$, and $A_{k-2} \to u_{k-1} u_k$. The A_i's are new variables. We replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \to u_i$

Example:

Convert the following CFG into Chomsky normal form

$$\begin{aligned} S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\lambda \end{aligned}$$

We start by adding a new start symbol

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\lambda \end{aligned}$$

The removing the λ-rule (in two steps)

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA|aB|a|SA|AS|S \\ A &\rightarrow B|S \\ B &\rightarrow b \end{aligned}$$

Removing the unit rules $S \to S$, then $S_0 \to S$ (in two steps) yields

$$\begin{aligned} S_0 &\to ASA|aB|a|SA|AS \\ S &\to ASA|aB|a|SA|AS \\ A &\to B|S \\ B &\to b \end{aligned}$$

Removing the unit rules $A \to B$ and $A \to S$ yields

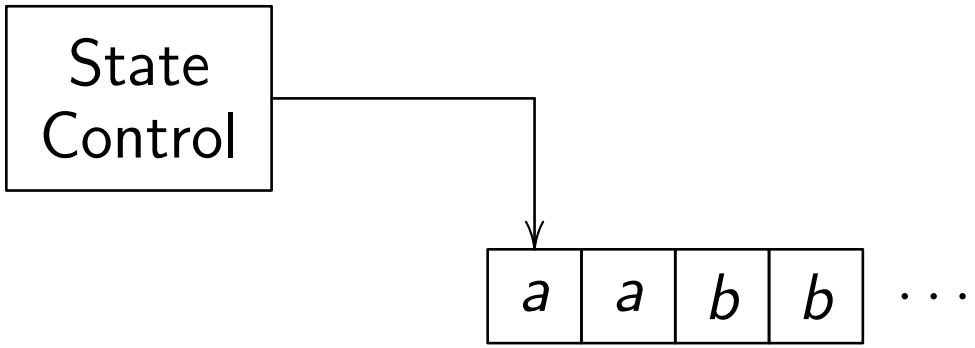
$$\begin{aligned} S_0 &\to ASA|aB|a|SA|AS \\ S &\to ASA|aB|a|SA|AS \\ A &\to b|ASA|aB|a|SA|AS \\ B &\to b \end{aligned}$$

Convert the remaining rules into the proper form by adding additional variables and rules: $A_1 \rightarrow SA$, and $U \rightarrow a$ gives:

$$
\begin{aligned}
S_0 &\rightarrow AA_1|UB|a|SA|AS \\
S &\rightarrow AA_1|UB|a|SA|AS \\
A &\rightarrow b|AA_1|UB|a|SA|AS \\
A_1 &\rightarrow SA \\
U &\rightarrow a \\
B &\rightarrow b
\end{aligned}
$$

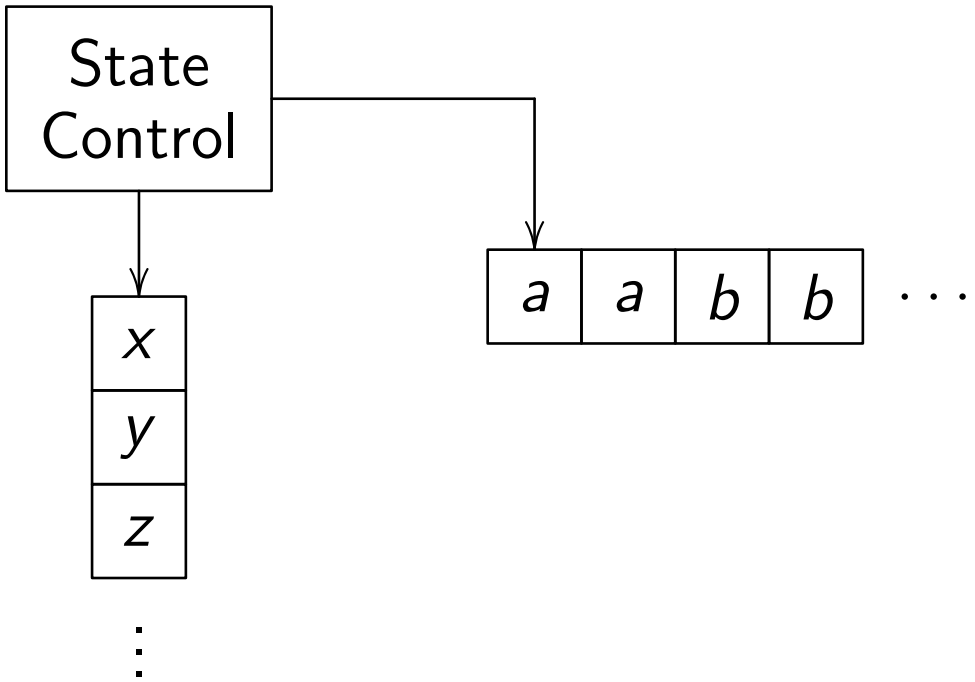
Automaton

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read



Pushdown Automaton

A pushdown automaton are like nondeterministic finite automata but have an extra component called a stack



- A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol pushes down all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up
- Writing a symbol on the stack is often referred to as pushing the symbol, and removing a symbol is referred to as popping it. Note that all access to the stack, for both reading and writing, may be done only at the top

Example

A finite automaton is unable to recognize the language

$$\{0^n 1^n; n \geq 0\}$$

because it cannot store very large numbers in its finite memory. A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If reading the input is finished exactly when the stack becomes empty of 0's, accept the input. If the stack becomes empty while 1s remain or if the 1's are finished while the stack still contains 0's or if any 0's appear in the input following 1's, reject the input

Formal definition of a Pushdown automata

A **nondeterministic pushdown (NPDA)** is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of states of the control unit
- Σ is a finite input alphabet
- Γ is a finite **stack alphabet**
- δ: $Q \times \Sigma_\lambda \times \Gamma_\lambda \mapsto \mathcal{P}(Q \times \Gamma^*)$ is the transition function
- q_0 is the initial state
- $F \subseteq Q$ is the set of final states

With $\Sigma_\lambda = \Sigma \cup \lambda$, and $\Gamma_\lambda = \Gamma \cup \lambda$

Pushdown automata computation

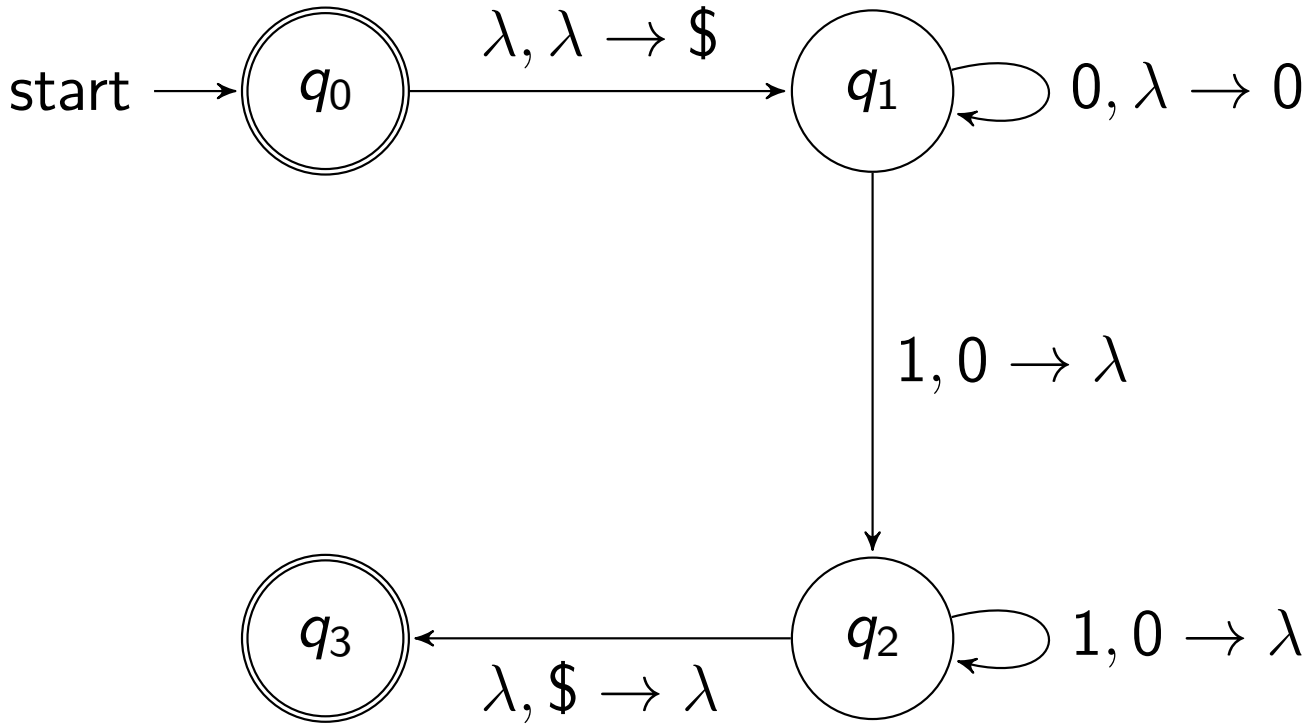
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\lambda$ and sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation

- $r_0 = q_0$ and $s_0 = \lambda$. This condition signifies that M starts out properly, in the start state and with an empty stack
- For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\lambda$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol
- $r_m \in F$. This condition states that an accept state occurs at the input end

State diagram

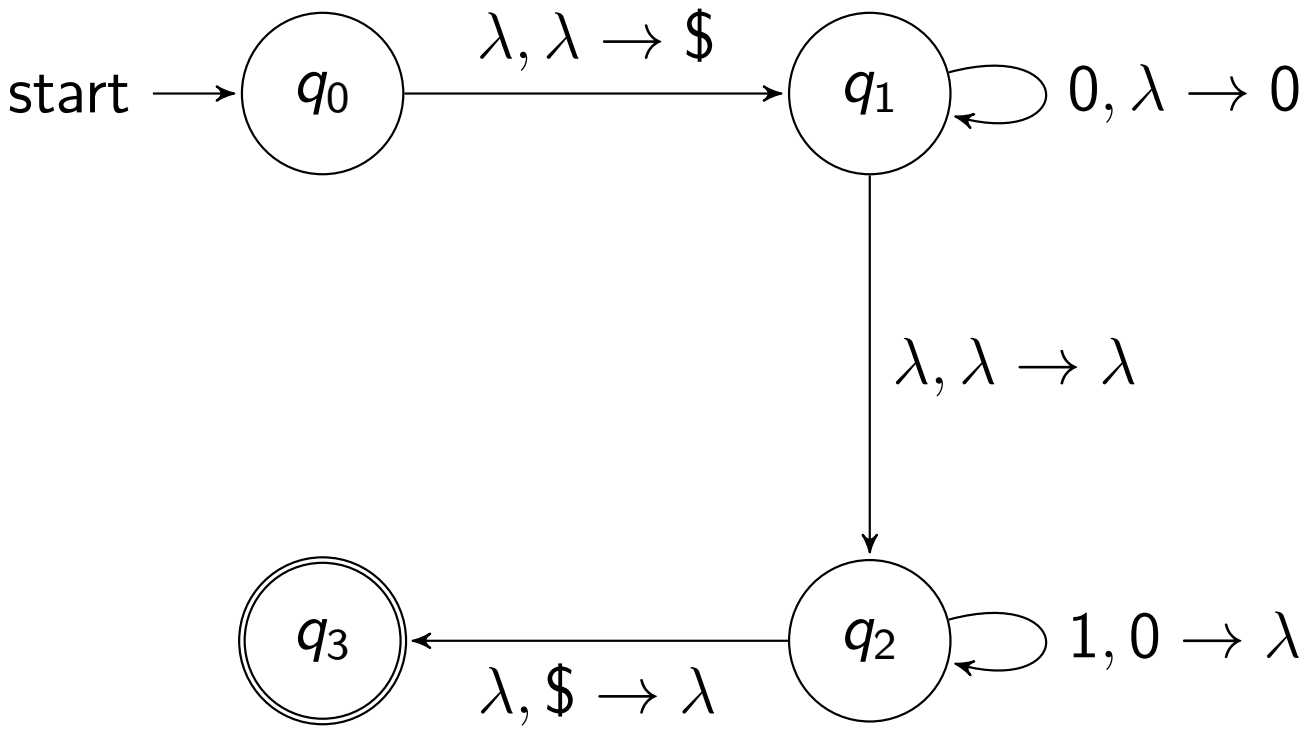
We write $a, b \rightarrow c$ to signify that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be λ. If a is λ, the machine may make this transition without reading any symbol from the input. If b is λ, the machine may make this transition without reading and popping any symbol from the stack. If c is λ, the machine does not write any symbol on the stack when going along this transition

State diagram 1

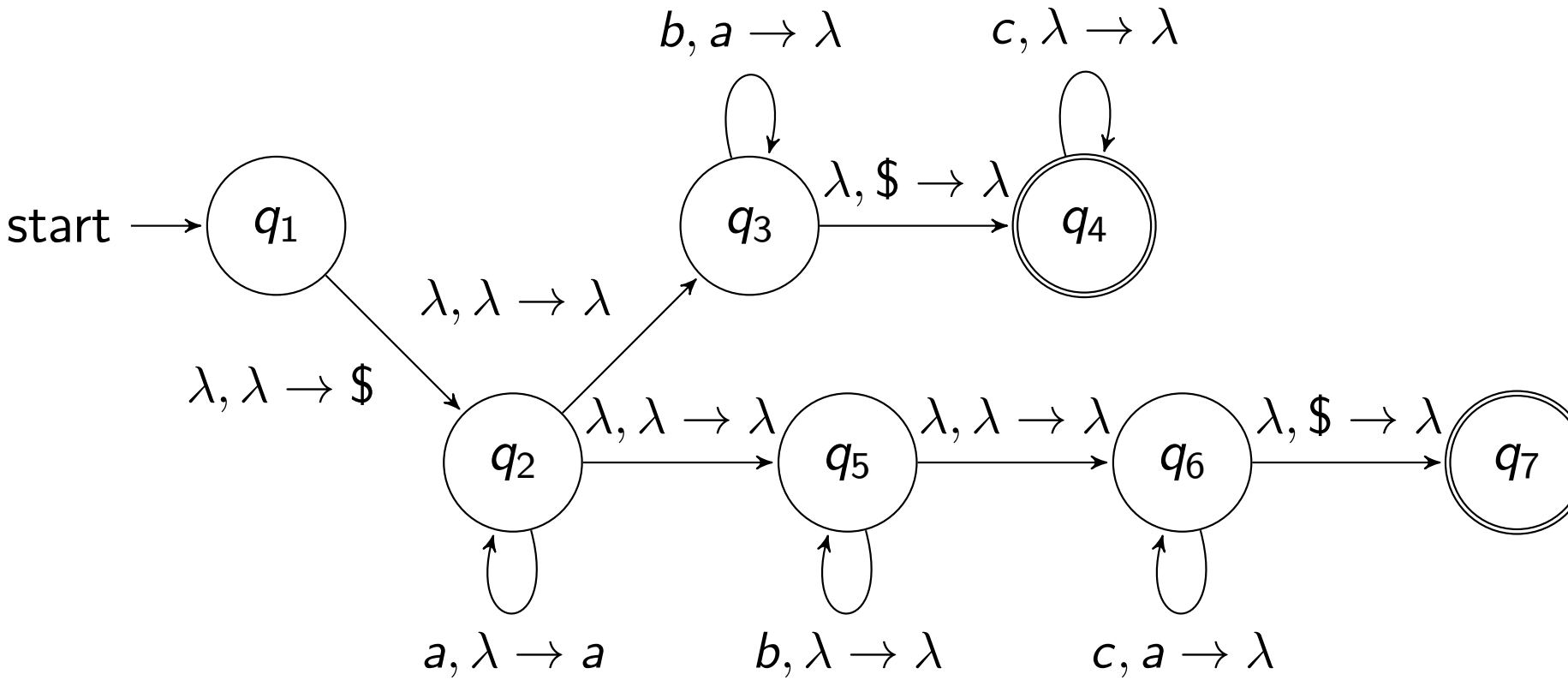


State diagram for the PDA M_1 that recognizes $\{0^n1^n; n \geq 0\}$

Or equivalently,

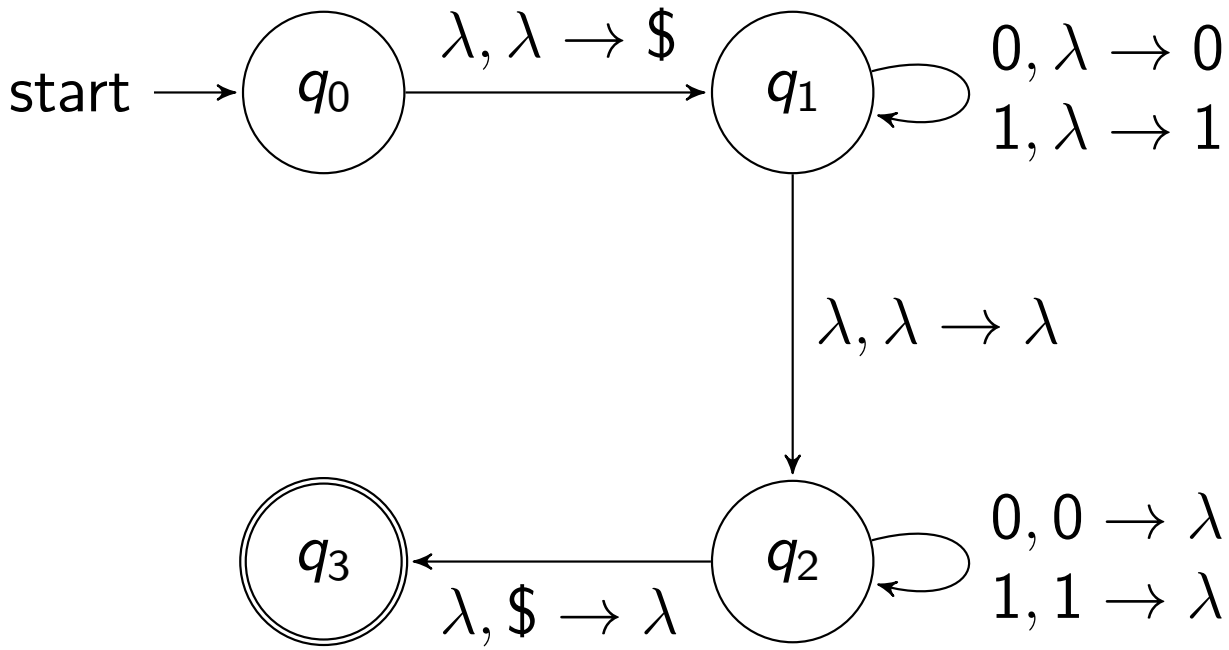


State diagram 2



State diagram for the PDA M_2 that recognizes $\{a^i b^j c^k; i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

State diagram 3



State diagram for the PDA M_3 that recognizes $\{ww^R; w \in \{0,1\}^*\}$

CFG and PDA equivalence

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

CFG to PDA conversion

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

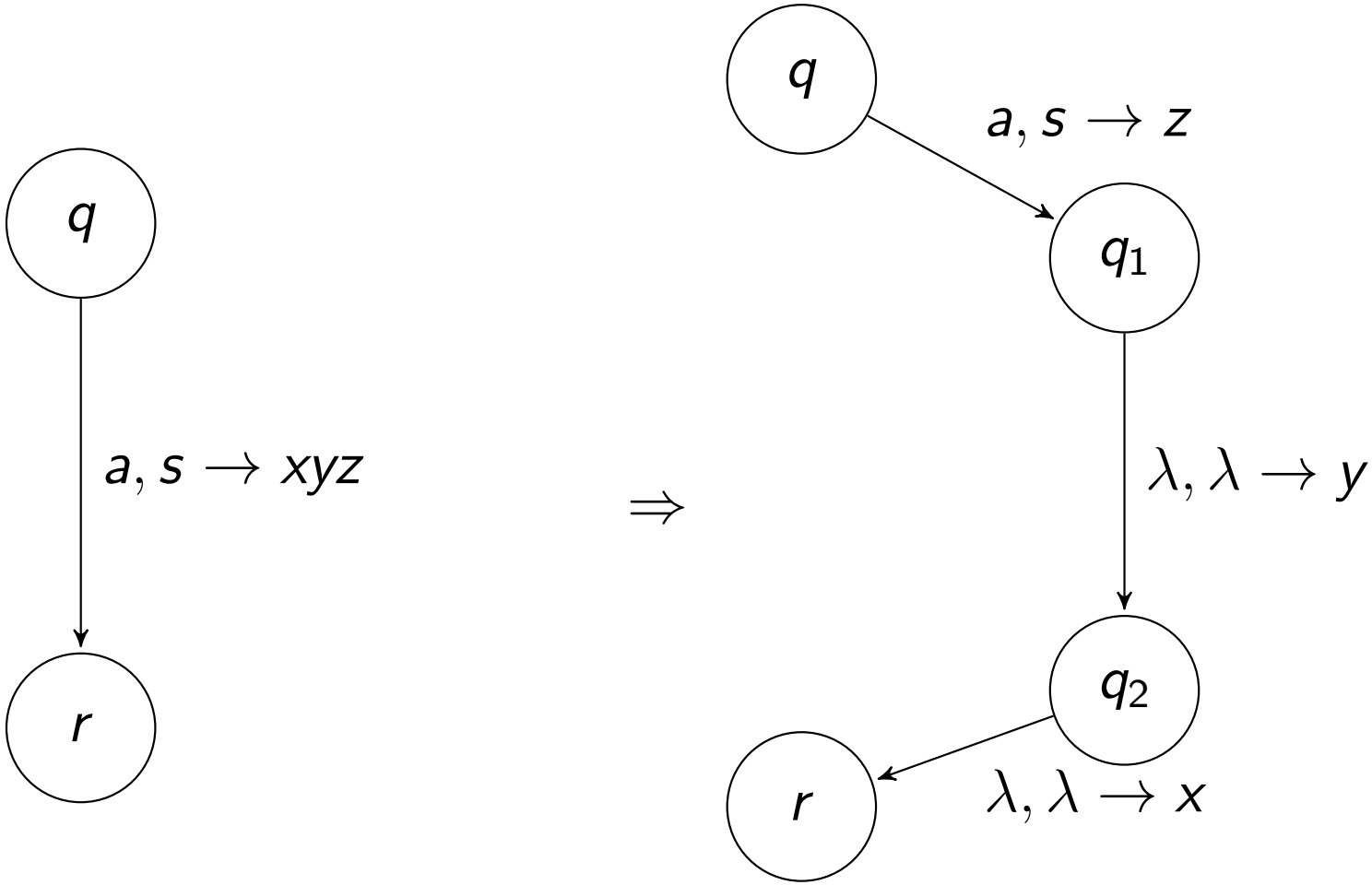
Proof idea:

The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another. Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string. Then P accepts if this string is identical to the string it has received as input

The following is an informal description of P:

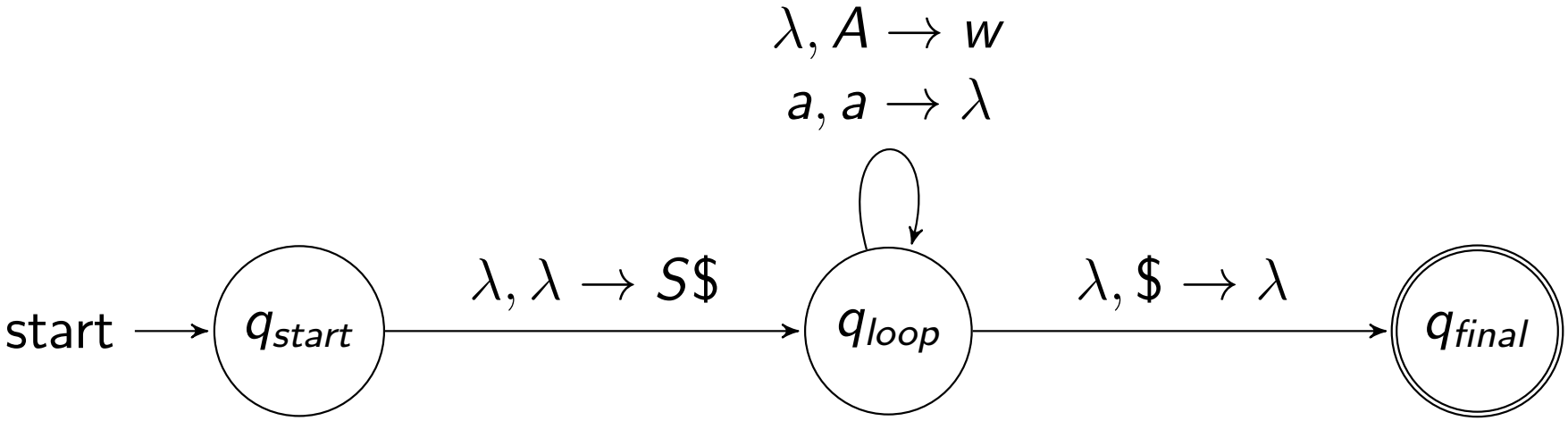
1) Place the marker symbol \$ and the start variable on the stack.
2) Repeat the following steps forever
   a) If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule
   b) If the top of stack is 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

The following notation is used to push a string into the stack



State diagram for P

The state diagram for a pushdown automata from a CFG is:



For every rule of the form $A \to w$, and a terminal

Example

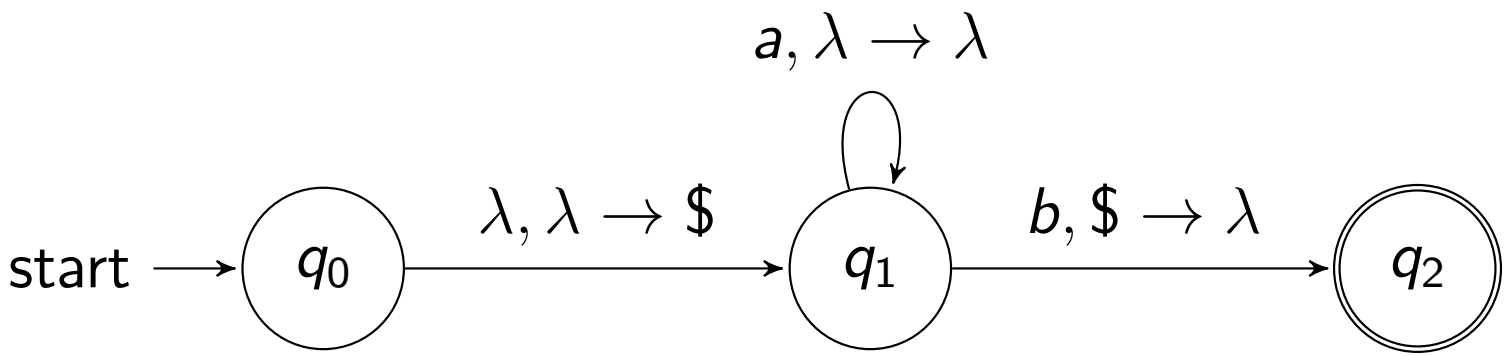
For the CFG G below:

$$\begin{aligned} S &\rightarrow aTb \mid b \\ T &\rightarrow Ta \mid \lambda \end{aligned}$$

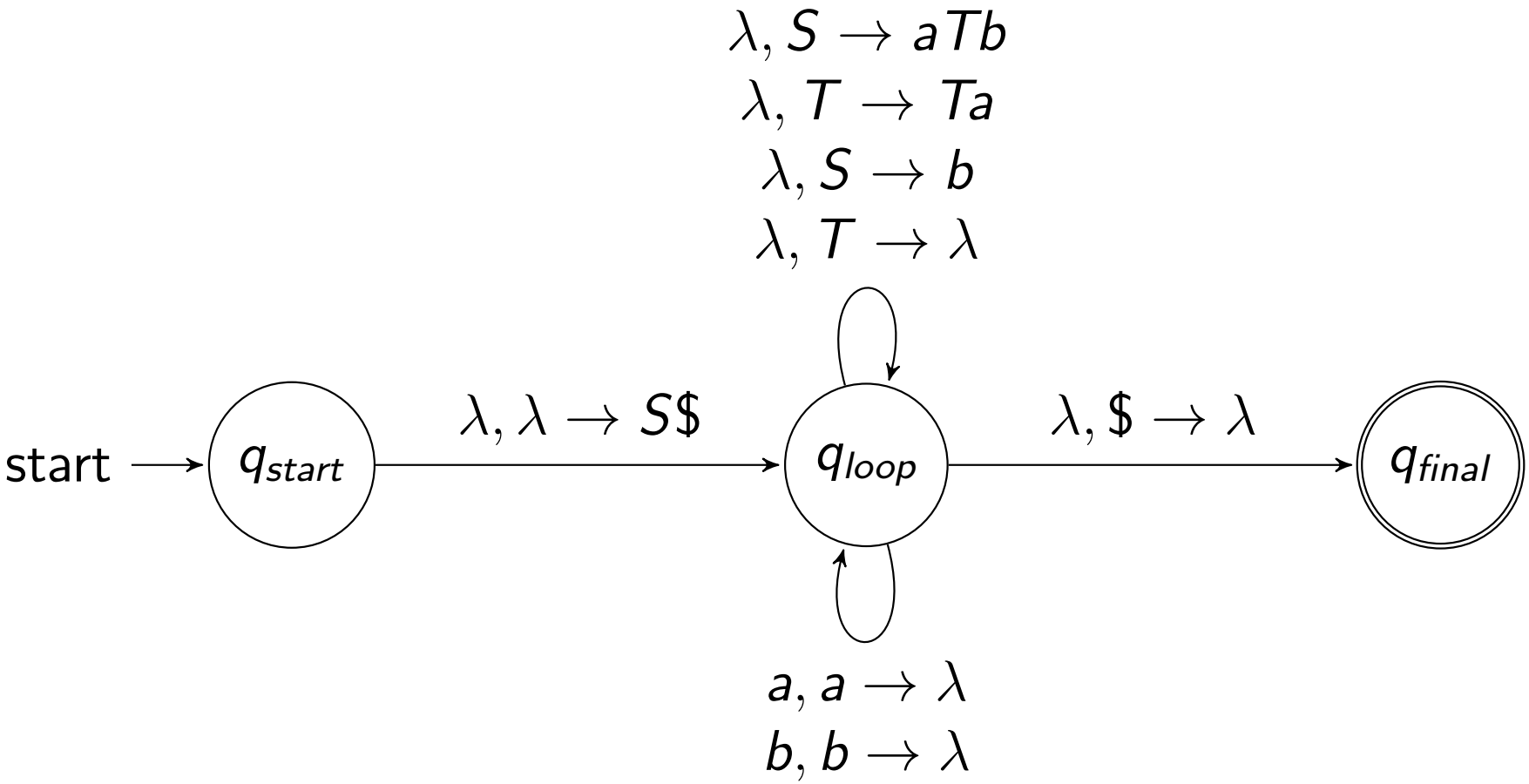
The language accepted is $L(G) = \ldots$?

PDA

A pushdown automata that recognizes $L(G)$:



The state diagram is:



Non-Context Free Languages

Pumping lemma for context-free languages:
If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

a) $uv^ixy^iz \in A$ for each $i \geq 0$

b) $|vy| > 0$

c) $|vxy| \leq p$

Show that the language $B = \{a^n b^n c^n; n \geq 0\}$ is not context free

Show that the language $C = \{a^i b^j c^k; 0 \leq i \leq j \leq k\}$ is not context free

Show that the language $D = \{ww; w \in \{0,1\}^*\}$ is not context free