Chapter 6: Pushdown Automata

Dr. Yuan Yao

University of Nottingham Ningbo China (UNNC)

Learning Outcomes

Learning outcomes

At the conclusion of this chapter, the students are expected to be able to:

- Describe the components of a nondeterministic pushdown automaton.
- State whether an input string is accepted by a nondeterministic pushdown automaton.
- Construct a pushdown automaton to accept a specific language.
- Given a context-free grammar in Greibach normal form, construct the corresponding pushdown automaton.
- Describe the differences between deterministic and nondeterministic pushdown automata.
- Describe the differences between deterministic and general context-free languages.

Introduction

Introduction

- We have seen that regular languages may be specified in different ways, depending on the application, e.g., using:
  - Regular Expression (for search and pattern matching)
  - Finite Automata (for computation)
  - Regular Grammars (for language design)
- We have seen that the above three are equivalent, and one may convert any one to the other two algorithmically.
- We entered the discussion of context-free languages with context-free grammars.
- It is natural to ask whether there are other ways of specifying CFLs.
- CFLs do not have anything similar to regular expressions.
- But they do have a machine model, i.e., pushdown automata.

Pushdown Automata

Pushdown Automata

- The key point why CFLs such as A^nB^n cannot be recognised by finite automata is that FA is restricted to have a finite amount of memories (i.e., the states).
- It indicates that recognition of a CFL may require storing an unbounded amount of information.

- Consider the following languages:
  - $A^nB^m = \{a^nb^m | n, m \geq 0\}$, which is regular.
  - $A^nB^n = \{a^nb^n | n \geq 0\}$, which is not regular.

- To recognise strings in A^nB^m, all we need to do is to check that $a's$ appear before b's.
- In contrast, for A^nB^n, we must also count the number of $a's$.
- Since n is unbounded, this counting cannot be done with a finite memory.

Pushdown Automata

- As the example of $A^n B^n$ shows, for the machine model of CFLs, we need a machine that can count without limit.
- But that is not enough. For instance, consider $WW^R = \{ww^R | w \in \{a, b\}^*\}$, which is a CFL.
- WW^R shows that we need more than unlimited counting ability:
  - We need the ability to store and match a sequence of symbols in reverse order.
- This suggests that we might try a stack as a storage mechanism, allowing unbounded storage that is restricted to operating like a stack.
- This gives us a class of machines called pushdown automata (PDA), which we use as a model of computation to process context-free languages.

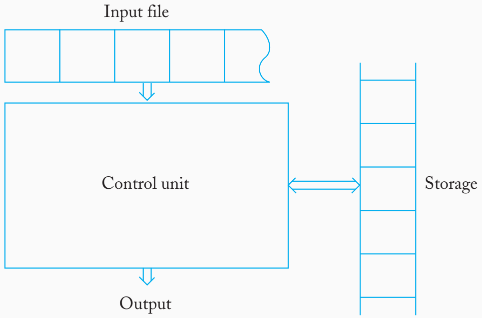
Pushdown Automata

- A pushdown automaton is essentially a finite automaton with a **stack** added as storage.
- As there is no limit on the size of the stack, pushdown automata do not have bounded memory limitation like finite automata.
- We will see that, pushdown automata are equivalent to CFGs, as long as we allow them to be nondeterministic.
- We can also define deterministic pushdown automata, but the language family associated with them is a proper subset of the context-free languages.
  - This is in contrast with the case of finite automata.
  - Remember that nondeterministic finite automata are equivalent to the deterministic ones.

Nondeterministic Pushdown Automata

Nondeterministic Pushdown Automata

- Each move of the control unit:
  - reads a symbol from the input file.
  - changes the contents of the stack (through the usual stack operations).
- Each move of the control unit is determined by:
  - the current input symbol.
  - and the symbol currently on top of the stack.
- The result of the move is a new state of the control unit and a change in the top of the stack.



Nondeterministic Pushdown Automata

- A *nondeterministic pushdown automaton (NPDA)* $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is defined by:
  - Q : the finite set of internal states of the control unit.
  - Σ : the finite set of input alphabet.
  - Γ : the finite set of stack alphabet.
  - δ : the transition function with the type signature:

    $$Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to P_f(Q \times \Gamma^*)$$

    where $P_f(Q \times \Gamma^*)$ is the set of *finite subsets* of $(Q \times \Gamma^*)$
  - $q_0 \in Q$: the initial state of the control unit.
  - $z \in \Gamma$: the stack start symbol.
  - $F \subseteq Q$: the set of final states.

Transition function δ

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to P_f(Q \times \Gamma^*)$$

- The arguments of δ are:
    - the current state of the control unit.
    - the current input symbol.
    - and the current symbol on top of the stack.
- The result is a finite set of pairs (q, x), where:
    - q is the next state of the control unit.
    - and x is a **string** that is put on top of the stack **in place of the single symbol** that was there before.

Transition function δ

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to P_f(Q \times \Gamma^*)$$

- λ-transition: the case where the second argument of δ is λ, i.e., a move that does not consume an input symbol.
- Note: δ always needs a stack symbol, i.e., no move is possible if the stack is empty.
- Finally, the requirement that the elements of the range of δ be a **finite** subset is necessary because:
    - $Q \times \Gamma^*$ is an infinite set and therefore has infinite subsets.
    - While an NPDA may have several choices for its moves, this choice must be restricted to a finite set of possibilities.

Sample NPDA Transitions

- Suppose the set of transition rules of an NPDA contains:

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}$$

  According to this rule, when:
  - the control unit is in state q_1
  - the input symbol is a
  - and the top of the stack is b

  then, one of two things can happen:
  1. the control unit goes into state q_2 and the string cd replaces b on top of the stack.
  2. the control unit goes into state q_3 with the symbol b removed from the top of the stack.
- Note: If a particular transition is not defined, the corresponding (state, symbol, stack top) configuration represents a dead state.

Exercise: A Sample NPDA

- Consider the following NPDA with:

$$Q = (\{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, \Gamma = \{0, 1\}, z = 0, F = \{q_3\})$$

with initial state q_0, and transition function δ given by:

$$\begin{aligned}
\delta(q_0, a, 0) &= \{(q_1, 10), (q_3, \lambda)\} \\
\delta(q_0, \lambda, 0) &= \{(q_3, \lambda)\} \\
\delta(q_1, a, 1) &= \{(q_1, 11)\} \\
\delta(q_1, b, 1) &= \{(q_2, \lambda)\} \\
\delta(q_2, b, 1) &= \{(q_2, \lambda)\} \\
\delta(q_2, \lambda, 0) &= \{(q_3, \lambda)\}
\end{aligned}$$

- Question: Find out the language accepted by this NPDA.

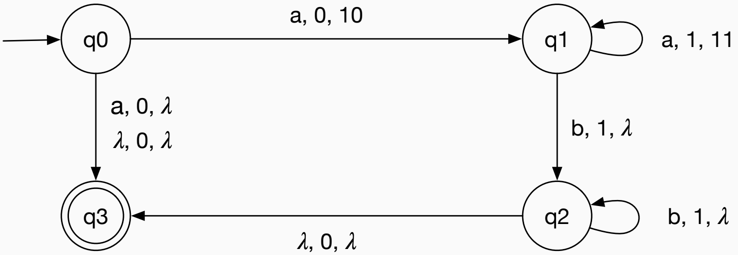
Exercise: A Sample NPDA

- From the initial state, if an a is read, the final state q_3 can be directly reached by removing a symbol 0 from the stack.

- As long as the control unit is in q_1, a 1 is pushed onto the stack when an a is read.
- The first b causes control to shift to q_2, which removes a symbol from the stack whenever a b is read.
- The NPDA will end in the (only) final state q_3 if and only if the input string is in the language $L = \{a^n b^n | n \geq 0\} \cup \{a\}$

Transition Graphs

- Similar to finite automata, we can also use transition graphs to represent NPDAs.
- In this representation we label the edges of the graph with three things:
    1. the current input symbol.
    2. the symbol at the top of the stack.
    3. and the string that replaces the top of the stack.
- The graph below represents the NPDA in the previous example.



Transition Graphs

- Draw a transition graph for the following NPDA:

$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, z\}, \delta, q_0, z, \{q_2\})$$

where δ is defined as follows:

$$\begin{aligned}
\delta(q_0, a, z) &= \{(q_1, a), (q_2, \lambda)\} \\
\delta(q_1, b, a) &= \{(q_1, b)\} \\
\delta(q_1, b, b) &= \{(q_1, b)\} \\
\delta(q_1, a, b) &= \{(q_2, \lambda)\}
\end{aligned}$$

Instantaneous Descriptions

- While transition graphs are convenient for describing NPDAs, they are not so suitable for formal reasoning.
- To trace the operation of an NPDA, we must keep track of:
  1. the current state of the control unit
  2. the unread part of the input string
  3. and the stack contents

- **Instantaneous Description**: The triplet (q, w, u) in which:
  1. q is the state of the control unit
  2. w is the unread part of the input string
  3. and u is the stack contents, with the leftmost symbol indicating the top of the stack

  is called an instantaneous description of a pushdown automaton.

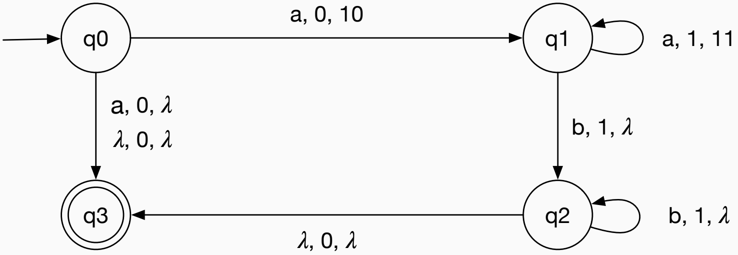
Instantaneous Descriptions

- A move from one instantaneous description to another will be denoted by the symbol $\vdash$ ($A \vdash B$ means B can be derived from A).
- We say $(q_1, aw, bx) \vdash (q_2, w, yx)$, if and only if:

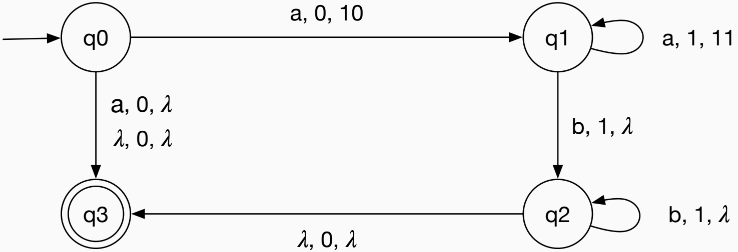
$$(q_2, y) \in \delta(q_1, a, b)$$

- A trace of the following NPDA with input string ab is:

$$(q_0, ab, 0) \vdash (q_1, b, 10) \vdash (q_2, \lambda, 0) \vdash (q_3, \lambda, \lambda)$$



Exercise: Instantaneous Description



- Write down two traces starting from $(q_0, abb, 0)$

The Language Accepted by an NPDA

- The language accepted by an NPDA is the set of all strings that cause the NPDA to halt in a final state, after starting in q_0 with only the stack symbol z on the stack.
- The final contents of the stack are irrelevant.
- As was the case with nondeterministic finite automata, the string is accepted if at least one of the computations cause the NPDA to halt in a final state.
- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be an NPDA. The language accepted by M is the set:

$$L(M) = \{w \in \Sigma^* | (q_0, w, z) \vdash^* (p, \lambda, u), p \in F, u \in \Gamma^*\}$$

Exercise: NPDA

- Draw a transition graph for the NPDA which accepts the following languages:

$$L = \{w \in \{a, b\}^* | n_a(w) = n_b(w)\}$$

  i.e., the strings which contains the same number $a's$ and $b's$.

Another example: WW^R

- Construct an NPDA to accept the language WW^R:

$$WW^R = \{ww^R | w \in \{a, b\}^+\}$$

Pushdown Automata and Context-Free Languages

NPDAs and CFLs

- So far, we have seen some example of CFLs and the NPDAs accepting them.
- In general, it seems that for every CFL L, there exists an NPDA M accepting it, i.e., $L(M) = L$.
- Conversely, for every NPDA M, the language $L(M)$ is context-free.

Greibach Normal Form (GNF)

- **Definition:** A CFL $G = (V, T, S, P)$ is said to be in *Greibach Normal Form* if all productions are in the following form:

$$A \to ax$$

  where $a \in T$ and $x \in V^*$.
- A CFL is in GNF if, in all of its productions, the right side consists of a single terminal symbol followed by any number of variables.
- The grammar below is in GNF:

$$\begin{aligned} S &\to aAB \mid bBB \mid bB \\ A &\to aA \mid bB \mid b \\ B &\to b \end{aligned}$$

Greibach Normal Form (GNF)

- **Theorem 6.7**: For every context-free grammar G there exists a grammar G' in GNF satisfying:
$$L(G') = L(G) - \{\lambda\}$$
- In general, the conversion of a given grammar to GNF, and the proof that this can always be done, are quite complicated. So, we do not present a proof here.

A Useful Substitution Rule

- **Theorem 6.1** Let $G = (V, T, S, P)$ be a context-free grammar. Suppose that P contains a production of the form:

  $$A \to x_1 B x_2$$

  Assume that A and B are different variables and that

  $$B \to y_1 \mid y_2 \mid \ldots \mid y_n$$

  is the set of all productions in P that have B as the left side. Let $G' = (V, S, T, P')$ be the grammar in which P' is constructed by replacing the above production for variable A by the following rule:

  $$A \to x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid \ldots \mid x_1 y_n x_2$$

  Then we have $L(G) = L(G')$

Greibach Normal Form (GNF)

- Practice: Write a grammar G_2 in GNF for the following grammar G_1, such that $L(G_1) = L(G_2)$:

$$S \rightarrow abSb \mid aa$$

NPDAs for CFLs

- **Theorem 7.1** For any given context-free language L, there exists an NPDA M such that:
$$L(M) = L$$
- By Theorem 6.7, for any CFL L, there exists a grammar $G = (V, T, S, P)$ in GNF, such that $L(G) = L$.
- The constructive proof of Theorem 7.1 provides an algorithm that can be used to build the corresponding NPDA, for any language specified by a grammar G in GNF.
- The resulting NPDA simulates grammar derivations by:
  - keeping variables on the stack.
  - while making sure that the input symbol matches the terminal on the right side of the production.

Sample Construction of an NPDA from a Grammar

- Consider the language generated by the grammar $S \to aSbb|a$.
- First of all, we need to convert the grammar to GNF:

$$
\begin{aligned}
S &\to aSA \,|\, a \\
A &\to bB \\
B &\to b
\end{aligned}
$$

- The corresponding NPDA has three states: $Q = \{q_0, q_1, q_f\}$, with q_0 as the initial state, and q_f as the only final state.
- How to generate productions?

Sample Construction of an NPDA from a Grammar

We then follow the procedure below to generate the productions:

- First, the start symbol S is placed on the stack with the following λ-transition:

  $$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

- The grammar productions are simulated with the transitions:

  $$\begin{aligned} \delta(q_1, a, S) &= \{(q_1, SA), (q_1, \lambda)\} \\ \delta(q_1, b, A) &= \{(q_1, B)\} \\ \delta(q_1, b, B) &= \{(q_1, \lambda)\} \end{aligned}$$

- When the stack start symbol z appears on top of the stack, the derivation is complete:

  $$\delta(q_1, \lambda, z) = \{(q_f, \lambda)\}$$

Construction of an NPDA from a Grammar

- The construction of the previous example can be generalised for any CFLs in GNF.

- For any CFLs $G = (V, T, S, P)$ in GNF, we could always construct a NPDA $M = (\{q_0, q_1, q_f\}, \Sigma, \Gamma, \delta, q_0, z, \{q_f\})$ such that:
- q_0 is the initial state and q_f is the only final state.
- $\Sigma = T$
- $\Gamma = V \cup \{z\}$ and $z \notin V$

Construction of an NPDA from a Grammar

The transition function δ is constructed as follows, which essentially carry out a leftmost derivation:

- A rule that pushes S on the stack and switches control to q_1 without consuming input:

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

- For every production of the form $A \rightarrow au$, we have

$$(q_1, u) \in \delta(q_1, a, A)$$

- A rule that switches the control unit to the final state q_f when there is no more input, and z appears at the top of the stack:

$$\delta(q_1, \lambda, z) = \{(q_f, z)\}$$

Construction of an NPDA from a Grammar

The transition function δ is constructed as follows, which essentially carry out a leftmost derivation:

- A rule that pushes S on the stack and switches control to q_1 without consuming input:
$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$
- For every production of the form $A \to au$, we have
$$(q_1, u) \in \delta(q_1, a, A)$$
- A rule that switches the control unit to the final state q_f when there is no more input, and z appears at the top of the stack:
$$\delta(q_1, \lambda, z) = \{(q_f, z)\}$$
- Finally, if $\lambda \in L$, we need to add the transition:$\delta(q_0, \lambda, z) = \{q_f, z\}$

Exercise

- Construct an NPDA M which accepts the languages generated by:

$$
\begin{aligned}
S &\rightarrow aA \\
A &\rightarrow aABC \mid bB \mid a \\
B &\rightarrow b \\
C &\rightarrow c
\end{aligned}
$$

- **Theorem 7.2:** If $L = L(M)$ for some NPDA M, then there exists a context-free grammar G, such that $L(G) = L(M)$. In other words, L is a context-free language.

- Proof: The basic idea behind the proof is to reverse the process in the proof of Theorem 7.1, i.e., to construct a grammar that simulates the moves of the NPDA M.
- In particular:
  - the content of the stack should be reflected in the variable part of sentential forms in derivations.
  - while the processed input is the terminal prefix of the sentential forms.

CFGs for NPDAs

- Nonetheless, the details of the proof are quite complicated.
  - For instance, the variables of the grammar should be defined not just based on the stack, but also the state of the machine.
  - Hence, we take triples $q_i A q_j$ where q_i and q_j are states, and A is a stack symbol, as the variables of the grammar.
- So, we do not present the proof here.

- Theorems 7.1 and 7.2 show that the following are equivalent:
  - Context-free Grammars
  - NPDAs
- Which specification one chooses depends on the purpose:
  - For specifying programming language constructs, grammars are more appropriate, as they are easier to understand by human beings.
  - For computational purposes (e.g., compilation of a program) the machine model, i.e., NPDA, is more appropriate.

Deterministic Pushdown Automata

Deterministic Pushdown Automata (DPDA)

- A deterministic pushdown accepter (DPDA) never has more than one choice in its moves.
- For every $q \in Q$, $a \in \Sigma \cup \{\lambda\}$ and $b \in \Gamma$:
  - $\delta(q, a, b)$ contains at most one element.
  - If $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every input symbol $c \in \Sigma$.
  - when a λ-move is possible for some configuration, no input-consuming alternative is available.

Deterministic Pushdown Automata (DPDA)

- Unlike the case for finite automata, a λ-transition does not necessarily mean the automaton is nondeterministic:
    - Since the top of the stack plays a role in determining the next move, the presence of λ-transitions does not automatically imply nondeterminism.

- Also, some transitions of a DPDA may be to the empty set, that is, undefined, so there may be dead configurations.
- This does not affect the definition either:
    - The only criterion for determinism is that, at all times, at most one possible move exists.

Differences betweent DFA and DPDA

- λ-transition
  - λ-transition is not allowed in DFA.
  - λ-transition is allowed in DPDA, but it rules out all other transitions (which does consume symbols).
- Total or partial function:
  - δ in a DFA must be a total function.
  - δ in a DPDA can be a partial function.
- Output of the function
  - The output of the λ in a DFA is a single state.
  - The output of the λ in a DPDA is a set which contains exactly one tuple from $Q \times \Gamma^*$

Example of a DPDA

Can you construct a DPDA for the following language?

$$A^n B^n = \{a^n b^n \mid n \geq 0\}$$

Deterministic Context-Free Languages (DCFLs)

- A context-free language L is deterministic if there is a DPDA M such that:

$$L = L(M)$$

- From the previous example, it is clear that $A^n B^n$ is a DCFL.
  - How to prove it?
- Another example is the language of marked palindromes over $\Sigma = \{a, b, x\}$ defined as:

$$WxW^R = \{ww^R | w \in \{a, b\}^*\}$$

- An intuitive reason is that, because the marker x does not appear in w or its reverse w^R, the machine can tell when it has reached the middle of the string.
- Practice: Construct a DPDA in JFLAP which accepts WxW^R.

Deterministic Context-Free Languages (DCFLs)

- There are, however, languages which are context-free, but not deterministic, i.e., deterministic and nondeterministic pushdown automata are not equivalent.
- For example, the language of unmarked (even length) palindromes over $\Sigma_1 = \{a, b\}$ is not deterministic:

$$WW^R = \{ww^R | w \in \{a, b\}^*\}$$

- An intuitive reason is as follows:
  - The PDA can read the input string only once, from left to right.
  - There is no marker telling the machine when it has reached the middle of the input string.
  - Hence, the machine must "guess" when it reaches the middle of the string.
- But this is not a proof. A correct proof needs a very clever argument.
  - For those who are interested, a proof that WW^R is not deterministic may be found in the John Martin textbook (Theorem 5.16, page 175)

Deterministic Context-Free Languages (DCFLs)

- There is another example, let $A^nB^{2n} = \{a^nb^{2n} | n \geq 0\}$ and define:

$$L = A^nB^n \cup A^nB^{2n}$$

- This language is context-free, but not deterministic.
- Why?

Deterministic Context-Free Languages (DCFLs)

- There is another example, let $A^n B^{2n} = \{a^n b^{2n} | n \geq 0\}$ and define:

$$L = A^n B^n \cup A^n B^{2n}$$

- This language is context-free, but not deterministic.
- An intuitive reason is as follows:
    - After encountering the first b, the PDA must "guess" whether to expect the string $a^n b^n$ or $a^n b^{2n}$.
    - but it cannot do both at the same time deterministically.
- Again, the proof is quite long and relies on the material that we have not discussed in our classes.
    - Thus, we do not discuss this proof here.
    - Those who are interested may read the proof in the Linz textbook.

Importance of DCFLs

- The importance of deterministic context-free languages lies in the fact that they can be **parsed efficiently**.
- Let us think of the pushdown automaton as a parsing device:
  - Since there is no backtracking involved, we can easily write a computer program for it, and we may expect that it will work efficiently.
  - Since there may be λ-transitions involved, we cannot immediately claim that this will yield a linear-time parser, but it puts us on the right track.
- **self-study (for those who are interested)**
  - **Section 7.4** of the Linz textbook presents a brief discussion of what grammars might be suitable for the description of deterministic context-free languages.
  - This is a topic that is crucial in the study of compilers, but not directly related to the LAC module.