COMP1600, week 9: Grammars and Push-down Automata (PDAs)

convenors: Dirk Pattinson, Pascal Bercher

lecturer: Pascal Bercher

slides based on those by: Dirk Pattinson
(with contributions by Victor Rivera and previous colleagues)

Semester 2, 2024

Australian
National
University

Overview of Week 9

- Introduction
- Grammars and Derivations
- The Chomsky Hierarchy
- Regular Grammars/Languages
- Context-Free Grammars/Languages
- Parse-trees and Ambiguity
- Pushdown Automata

Introduction

Terminology Recap: Formal Languages

- The **alphabet** or **vocabulary** of a formal language is a set of **tokens** (or **letters**). It is usually denoted Σ.
- A **string** over Σ is a *sequence* of tokens.
    - e.g., sequence may be empty, giving empty string ϵ
    - e.g., $ababc$ is a string over $\Sigma = \{a, b, c\}$

Terminology Recap: Formal Languages

- The **alphabet** or **vocabulary** of a formal language is a set of **tokens** (or **letters**). It is usually denoted Σ.
- A **string** over Σ is a *sequence* of tokens.
  - e.g., sequence may be empty, giving empty string ϵ
  - e.g., *ababc* is a string over $\Sigma = \{a, b, c\}$
- A **language** with alphabet Σ is some set of strings over Σ.
  - e.g., the set of all strings Σ^*
  - e.g., the set of all strings of even length, $\{w \in \Sigma^* \mid |w| \ \% \ 2 \ == 0\}$
  - e.g., ... we had many examples in the last two weeks!
  - e.g., *any* subset of Σ^*

Specifying Languages

Languages can be given ...

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$

Specifying Languages

Languages can be given . . .

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.

Specifying Languages

Languages can be given . . .

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.
- algebraically by regular expressions, e.g. $L = L(r)$ for RegEx r

Specifying Languages

Languages can be given …

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.
- algebraically by regular expressions, e.g. $L = L(r)$ for RegEx r
- by an automaton, e.g. $L = L(A)$ for some FSA A (we had 3 kinds!)

Specifying Languages

Languages can be given ...

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.
- algebraically by regular expressions, e.g. $L = L(r)$ for RegEx r
- by an automaton, e.g. $L = L(A)$ for some FSA A (we had 3 kinds!)
- *by a grammar* (this lecture)

Specifying Languages

Languages can be given ...

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.
- algebraically by regular expressions, e.g. $L = L(r)$ for RegEx r
- by an automaton, e.g. $L = L(A)$ for some FSA A (we had 3 kinds!)
- *by a grammar* (this lecture)

Grammar.

- a concept that has been invented in linguistics to describe natural languages
- describes how strings are *constructed* rather than how membership can be *checked* (e.g., by an automaton; though does it? what does a RegEx do?)
- *the* main tool to describe syntax; grammars are extremely important!

Specifying Languages

Languages can be given . . .

- as a finite enumeration, e.g. $L = \{\epsilon, a, ab, abb\}$
- as a set, by giving an acceptance predicate, i.e., $L = \{w \in \Sigma^* \mid P(w)\}$ for some alphabet Σ. E.g., $P(w) = |w| \ \% \ 2 \ == 0$.
- algebraically by regular expressions, e.g. $L = L(r)$ for RegEx r
- by an automaton, e.g. $L = L(A)$ for some FSA A (we had 3 kinds!)
- *by a grammar* (this lecture)

Grammar.

- a concept that has been invented in linguistics to describe natural languages
- describes how strings are *constructed* rather than how membership can be *checked* (e.g., by an automaton; though does it? what does a RegEx do?)
- *the* main tool to describe syntax; grammars are extremely important!
- *Fun fact:* My expertise on Hierarchical Planning (a subfield of Artificial Intelligence) largely overlaps with formal grammars!

Grammars and Derivations

Formal Grammars

Formal Definition. A *grammar* is a quadruple $\langle V_t, V_n, S, P\rangle$ where

- V_t is a finite set of *terminal symbols* (the alphabet)
- V_n is a finite set of **non-terminal symbols** disjoint from V_t (Notation: $V = V_t \cup V_n$)
- S is a distinguished non-terminal symbol called the *start symbol*
- P is a set of *productions* (also called *production rules*), each written

$$\alpha \to \beta$$

  where
  - $\alpha \in V^* V_n V^*$ (i.e., at least one non-terminal in α)
  - $\beta \in V^*$ (i.e., β is *any* sequence of symbols)

Example (for Syntax)

The grammar (recall: grammars have the form $\langle V_t, V_n, S, P \rangle$)

$$G = \langle \{a, b\},\ \{S, A\},\ S,\ \{S \rightarrow aAb,\ aA \rightarrow aaAb,\ A \rightarrow \epsilon\} \rangle$$

has the following components:

Example (for Syntax)

The grammar (recall: grammars have the form $\langle V_t, V_n, S, P \rangle$)

$$G = \langle \{a, b\},\ \{S, A\},\ S,\ \{S \to aAb,\ aA \to aaAb,\ A \to \epsilon\} \rangle$$

has the following components:

- *Terminals:* $\{a, b\}$
- *Non-terminals:* $\{S, A\}$
- *Start symbol:* S
- *Productions:*

$$\begin{aligned} S &\to aAb \\ aA &\to aaAb \\ A &\to \epsilon \end{aligned}$$

Example (for Syntax)

The grammar (recall: grammars have the form $\langle V_t, V_n, S, P \rangle$)

$$G = \langle \{a, b\},\ \{S, A\},\ S,\ \{S \to aAb,\ aA \to aaAb,\ A \to \epsilon\} \rangle$$

has the following components:

- *Terminals:* $\{a, b\}$
- *Non-terminals:* $\{S, A\}$
- *Start symbol:* S
- *Productions:*

$$S \to aAb$$
$$aA \to aaAb$$
$$A \to \epsilon$$

Notation.

- Often, we just list the productions P, as all other components can be inferred (S is the standard notation for the start symbol)
- The notation $\alpha \to \beta_1 \mid \cdots \mid \beta_n$ abbreviates the *set* of productions

$$\alpha \to \beta_1, \quad \alpha \to \beta_2, \quad \ldots, \quad \alpha \to \beta_n$$

(like for inductive data types)

Derivations

Intuition.

- A production $\alpha \rightarrow \beta$ tells you what you can "make" if you have α: you can turn it into β. It allows us to re-write any string $\gamma\alpha\rho$ to $\gamma\beta\rho$.
- Notation: $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

Derivations

Intuition.

- A production $\alpha \to \beta$ tells you what you can "make" if you have α: you can turn it into β. It allows us to re-write any string $\gamma\alpha\rho$ to $\gamma\beta\rho$.
- Notation: $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

Derivations.

- $\alpha \Rightarrow^* \beta$ if α can be re-written to β in 0 or more steps
- so $\Rightarrow^*$ is the *reflexive transitive closure* of $\Rightarrow$.

Derivations

Intuition.

- A production $\alpha \to \beta$ tells you what you can "make" if you have α: you can turn it into β. It allows us to re-write any string $\gamma\alpha\rho$ to $\gamma\beta\rho$.
- Notation: $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

Derivations.

- $\alpha \Rightarrow^* \beta$ if α can be re-written to β in 0 or more steps
- so $\Rightarrow^*$ is the *reflexive transitive closure* of $\Rightarrow$.

Sentential Forms of a grammar.

- informally: all strings from $(V_t \cup V_n)^*$ that can be generated from S
- formally: $S(G) = \{w \in V^* \mid S \Rightarrow^* w\}$.

Derivations

Intuition.

- A production $\alpha \to \beta$ tells you what you can "make" if you have α: you can turn it into β. It allows us to re-write any string $\gamma\alpha\rho$ to $\gamma\beta\rho$.
- Notation: $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

Derivations.

- $\alpha \Rightarrow^* \beta$ if α can be re-written to β in 0 or more steps
- so $\Rightarrow^*$ is the *reflexive transitive closure* of $\Rightarrow$.

Sentential Forms of a grammar.

- informally: all strings from $(V_t \cup V_n)^*$ that can be generated from S
- formally: $S(G) = \{w \in V^* \mid S \Rightarrow^* w\}$.

Language of a grammar.

- informally: all strings of terminal symbols that can be generated from the start symbol S
- formally: $L(G) = \{w \in V_t^* \mid S \Rightarrow^* w\}$, which is a subset of $S(G)$
- that's the same as $L(G) = S(G) \cap V_t^*$

Example

Productions of the grammar G.

$$S \to aAb, \qquad aA \to aaAb, \qquad A \to \epsilon.$$

Example Derivation.

Example

Productions of the grammar G.

$$S \rightarrow aAb, \qquad aA \rightarrow aaAb, \qquad A \rightarrow \epsilon.$$

Example Derivation.

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

- last string $aaabbb$ is a *word*, others are *sentential forms*

Example

Productions of the grammar G.

$$S \to aAb, \qquad aA \to aaAb, \qquad A \to \epsilon.$$

Example Derivation.

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

- last string $aaabbb$ is a *word*, others are *sentential forms*

Language of grammar G.

Example

Productions of the grammar G.

$$S \to aAb, \qquad aA \to aaAb, \qquad A \to \epsilon.$$

Example Derivation.

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

- last string $aaabbb$ is a *word*, others are *sentential forms*

Language of grammar G.

$$L(G) = \{a^n b^n \mid n \in \mathbb{N},\ n \geq 1\}$$

Alternative Grammar for the *same* language

Example

Productions of the grammar G.

$$S \to aAb, \qquad aA \to aaAb, \qquad A \to \epsilon.$$

Example Derivation.

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

- last string $aaabbb$ is a *word*, others are *sentential forms*

Language of grammar G.

$$L(G) = \{a^n b^n \mid n \in \mathbb{N}, \, n \geq 1\}$$

Alternative Grammar for the *same* language

$$S \to aSb, \qquad S \to ab.$$

The Chomsky Hierarchy

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

Context-free: (type 2) the left of each production must be a *single non-terminal*.

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

Context-free: (type 2) the left of each production must be a *single non-terminal*. (**Q.** You *might* think that's not type 1, why?)

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

Context-free: (type 2) the left of each production must be a *single non-terminal*. (**Q.** You *might* think that's not type 1, why?)

Regular: (type 3) As for type 2, but the right of each production is further constrained (details to come).

The Chomsky Hierarchy (by Noam Chomsky, a Linguist!)

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

Context-free: (type 2) the left of each production must be a *single non-terminal*. (**Q.** You *might* think that's not type 1, why?)

Regular: (type 3) As for type 2, but the right of each production is further constrained (details to come).

This also gives us a way to classify *languages*. (Next slide.)

Classification of Languages

Definition. A language is *type n* if it can be generated by a type n grammar.

Immediate Fact.

- Every language of type $n + 1$ is also of type n.
- E.g., every context-free language (type 2) is also context-sensitive (type 1).

Classification of Languages

Definition. A language is *type n* if it can be generated by a type n grammar.

Immediate Fact.

- Every language of type $n + 1$ is also of type n.
- E.g., every context-free language (type 2) is also context-sensitive (type 1).

Establishing that a *language* is of type n

- give a grammar of type n that generates the language
- usually the easier task (and often fun! like designing an automaton/RegEx)

Classification of Languages

Definition. A language is *type n* if it can be generated by a type n grammar.

Immediate Fact.

- Every language of type $n+1$ is also of type n.
- E.g., every context-free language (type 2) is also context-sensitive (type 1).

Establishing that a *language* is of type n

- give a grammar of type n that generates the language
- usually the easier task (and often fun! like designing an automaton/RegEx)

Disproving that a language is of type n

- must show that *no* type n-grammar generates the language
- usually a *difficult* problem. (There are complex theorems to help.)

Example – Language $\{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$

Different grammars for this language

- *Unrestricted (type 0):*

$$\begin{aligned} S &\to aAb \\ aA &\to aaAb \\ A &\to \epsilon \end{aligned}$$

- *Context-free (type 2):*

$$\begin{aligned} S &\to ab \\ S &\to aSb \end{aligned}$$

Recall. We know from last week that there is no DFA accepting L

- We will see that this means that there's no regular grammar
- so the language is context-free, but not regular.

Regular (Type 3) Grammars

Definition. A grammar is *regular* if all its productions are either *right-linear*, i.e. of the form

$$A \to aB \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon$$

or *left-linear*, i.e., of the form

$$A \to Ba \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon.$$

Regular (Type 3) Grammars

Definition. A grammar is *regular* if all its productions are either *right-linear*, i.e. of the form

$$A \to aB \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon$$

or *left-linear*, i.e., of the form

$$A \to Ba \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon.$$

- right and left linear grammars are equivalent: they generate the same languages (and can hence be turned into each other)
- we focus on *right linear* ones (it's probably slightly more intuitive)
- i.e., one symbol is *generated* at a time (cf. DFA/NFA!)
- rule application terminates with terminal symbols or ϵ

Regular (Type 3) Grammars

Definition. A grammar is *regular* if all its productions are either *right-linear*, i.e. of the form

$$A \to aB \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon$$

or *left-linear*, i.e., of the form

$$A \to Ba \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \epsilon.$$

- right and left linear grammars are equivalent: they generate the same languages (and can hence be turned into each other)
- we focus on *right linear* ones (it's probably slightly more intuitive)
- i.e., one symbol is *generated* at a time (cf. DFA/NFA!)
- rule application terminates with terminal symbols or ϵ

Next Goal. Relate regular languages to DFAs/NFAs/RegExs.

Regular Grammars/Languages

Regular Languages – Many Views

Theorem. Let L be a language. Then the following are equivalent:

- L is the language generated by a *right-linear grammar*;
- L is the language generated by a *left-linear grammar*;
- L is the language accepted by some *DFA*;
- L is the language accepted by some *NFA*;
- L is the language specified by a *regular expression*.

Regular Languages – Many Views

Theorem. Let L be a language. Then the following are equivalent:

- L is the language generated by a *right-linear grammar*;
- L is the language generated by a *left-linear grammar*;
- L is the language accepted by some *DFA*;
- L is the language accepted by some *NFA*;
- L is the language specified by a *regular expression*.

So far.

- have seen that NFAs and DFAs generate the same languages
- have shown that regular expressions can be turned into NFAs (hence DFAs)
- claimed that DFAs can be turned into regular expressions

Goal.

Regular Languages – Many Views

Theorem. Let L be a language. Then the following are equivalent:

- L is the language generated by a *right-linear grammar*;
- L is the language generated by a *left-linear grammar*;
- L is the language accepted by some *DFA*;
- L is the language accepted by some *NFA*;
- L is the language specified by a *regular expression*.

So far.

- have seen that NFAs and DFAs generate the same languages
- have shown that regular expressions can be turned into NFAs (hence DFAs)
- claimed that DFAs can be turned into regular expressions

Goal. Show that NFAs and right-linear grammars generate the same languages.

From NFAs to Right-linear Grammars

Given. Take an NFA $A = (\Sigma, S, s_0, F, R)$.

- alphabet, state set, initial state, final states, transition relation

Construction of a right-linear grammar

- *terminal symbols* are elements of the alphabet Σ;
- *non-terminal symbols* are the states S;
- *start symbol* is the start state s_0;
- *productions* are constructed as follows:

From NFAs to Right-linear Grammars

Given. Take an NFA $A = (\Sigma, S, s_0, F, R)$.

- alphabet, state set, initial state, final states, transition relation

Construction of a right-linear grammar

- *terminal symbols* are elements of the alphabet Σ;
- *non-terminal symbols* are the states S;
- *start symbol* is the start state s_0;
- *productions* are constructed as follows:

Each *transition*	gives *production*
$T \xrightarrow{a} U$	$T \rightarrow aU$
Each *final state*	gives *production*
$T \in F$	$T \rightarrow \epsilon$

(Formally, a transition $T \xrightarrow{a} U$ means $(T, a, U) \in R$.)

Observation.

From NFAs to Right-linear Grammars

Given. Take an NFA $A = (\Sigma, S, s_0, F, R)$.

- alphabet, state set, initial state, final states, transition relation

Construction of a right-linear grammar

- *terminal symbols* are elements of the alphabet Σ;
- *non-terminal symbols* are the states S;
- *start symbol* is the start state s_0;
- *productions* are constructed as follows:

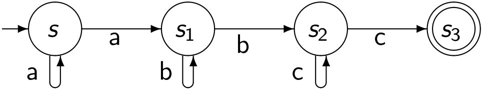
Each *transition*	gives *production*
$T \xrightarrow{a} U$	$T \to aU$
Each *final state*	gives *production*
$T \in F$	$T \to \epsilon$

(Formally, a transition $T \xrightarrow{a} U$ means $(T, a, U) \in R$.)

Observation. The grammar so generated is right-linear, and hence regular.

NFAs to Right-linear Grammars – Example

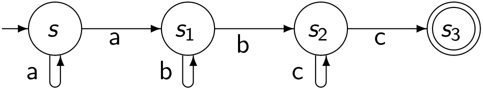
Given. A non-deterministic automaton



Equivalent Grammar obtained by construction

NFAs to Right-linear Grammars – Example

Given. A non-deterministic automaton



Equivalent Grammar obtained by construction

$$S \to aS \qquad S_2 \to cS_2$$
$$S \to aS_1 \qquad S_2 \to cS_3$$
$$S_1 \to bS_1 \qquad S_3 \to \varepsilon$$
$$S_1 \to bS_2$$

(We capitalised the NFA states to make clear they are non-terminals!)

Exercise. Convince yourself that the NFA accepts precisely the words that the grammar generates. (We still owe a proof of correctness of the translation.)

From Right-linear Grammars to NFAs

Given. Right-linear grammar (V_t, V_n, S, P)

- terminals, non-terminals, start symbol, productions

Construction of an equivalent NFA has

- *alphabet* is the terminal symbols V_t;
- *states* are the *non-terminal symbols* V_n plus new state S_f (for final);
- *start state* is the start symbol S;
- *final states* are S_f and *all* non-terminals T such that there exists a production $T \rightarrow \varepsilon$;
- *transition relation* is constructed as follows:

Each *production*	gives *transition*
$T \rightarrow aU$	$T \xrightarrow{a} U$
$T \rightarrow a$	$T \xrightarrow{a} S_f$

Right-linear Grammars to NFAs – Ex.

Given. Grammar G with the productions

$$S \to 0 \qquad S \to 1T$$

$$T \to \varepsilon \qquad T \to 0T \qquad T \to 1T$$

(generates the language

Right-linear Grammars to NFAs – Ex.

Given. Grammar G with the productions

$$S \to 0 \qquad S \to 1T$$
$$T \to \varepsilon \qquad T \to 0T \qquad T \to 1T$$

(generates the language of binary strings without leading zeros)

Equivalent Automaton obtained by construction.

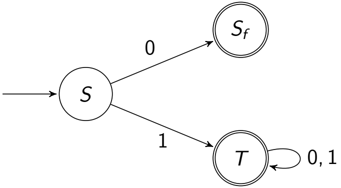
Right-linear Grammars to NFAs – Ex.

Given. Grammar G with the productions

$$S \to 0 \qquad S \to 1T$$
$$T \to \varepsilon \qquad T \to 0T \qquad T \to 1T$$

(generates the language of binary strings without leading zeros)

Equivalent Automaton obtained by construction.



Exercise. Convince yourself that the NFA accepts precisely the words that the grammar generates. (Again we still owe a correctness proof of the translation.)

Context-Free Grammars/Languages

Context-Free (Type 2) Grammars (CFGs)

Recall. A grammar is type-2 or *context-free* if all productions have the form

$$A \to w$$

where $A \in V_n$ is a non-terminal, and $w \in V^*$ is an (arbitrary) string.

- left side is non-terminal
- right side can be anything
- *independent* of context, replace LHS (left hand side) with RHS (right HS).

In Contrast. Context-Sensitive grammars may have productions

$$\alpha A \beta \to \alpha w \beta$$

which may only replace A by w if A appears in context α_β

Example

Goal. Design a CFG for the language

$$L = \{a^m b^n c^{m-n} \mid m \geq n \geq 0\}$$

Strategy. First, understand the language!

Example

Goal. Design a CFG for the language

$$L = \{a^m b^n c^{m-n} \mid m \geq n \geq 0\}$$

Strategy. First, understand the language! Every word $w \in L$ can be split

$$w = a^{m-n} a^n b^n c^{m-n}$$

and hence $L = \{a^k a^n b^n c^k \mid n, k \geq 0\}$

Resulting Grammar.

Example

Goal. Design a CFG for the language

$$L = \{a^m b^n c^{m-n} \mid m \geq n \geq 0\}$$

Strategy. First, understand the language! Every word $w \in L$ can be split

$$w = a^{m-n} a^n b^n c^{m-n}$$

and hence $L = \{a^k a^n b^n c^k \mid n, k \geq 0\}$

- convenient to *not* have comparison between n and m
- generate $a^k \dots c^k$, i.e., same number of leading as and trailing c^s
- fill $\dots$ in the middle by $a^n b^n$, i.e., same number of as and bs
- use different non-terminals for both phases of the construction

Resulting Grammar. (productions only)

$$\begin{aligned} S &\rightarrow aSc \mid T \\ T &\rightarrow aTb \mid \epsilon \end{aligned}$$

Example cont'd

Grammar

$$\begin{aligned} S &\rightarrow aSc \mid T \\ T &\rightarrow aTb \mid \epsilon \end{aligned}$$

Example Derivation. of $aaabbc$:

$$\begin{aligned} S &\Rightarrow aSc \\ &\Rightarrow aTc \\ &\Rightarrow aaTbc \\ &\Rightarrow aaaTbbc \\ &\Rightarrow aaabbc \end{aligned}$$

The Power of Context-Free Grammars

A fun example (for the “usefulness” of CFGs):
`http://pdos.csail.mit.edu/scigen`

This tool generates (fake) scientific papers based on formal grammars!

Note that:

- Sadly, the generator doesn’t seem to work anymore.
- But the page mentions where their (fake papers) were accepted!
- They link similar “services”.
- Maybe you can still find a working version somewhere...

Parse Trees and Ambiguity

Parse Trees

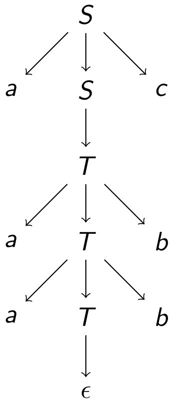
Idea. Represent derivation as *tree* rather than as list of rule applications

- describes where and how productions have been applied
- generated word can be collected at the leaves

Example for the grammar that we have just constructed

$$S \rightarrow aSc \mid T$$
$$T \rightarrow aTb \mid \epsilon$$

word: $w = aaabbc$



Parse Trees Carry Semantics

Take the code

```
if e1 then if e2 then s1 else s2
```

where **e1**, **e2** are boolean expressions and `s1`, `s2` are subprograms.

Two Readings.

```
if e1 then ( if e2 then s1 else s2 )
```

and

```
if e1 then ( if e2 then s1 ) else s2
```

Goal. *unambiguous* interpretation of the code leading to *determined* and *clear* program execution.

Ambiguity

Recall that we can present CFG derivations as *parse trees*.

Until now this was merely a pretty presentation; now it will become important.

Definitions:

- A context-free grammar G is **unambiguous** iff every string can be derived by ***at most*** one parse tree.
- G is **ambiguous** iff there exists a word $w \in L(G)$ derivable by more than one parse tree.

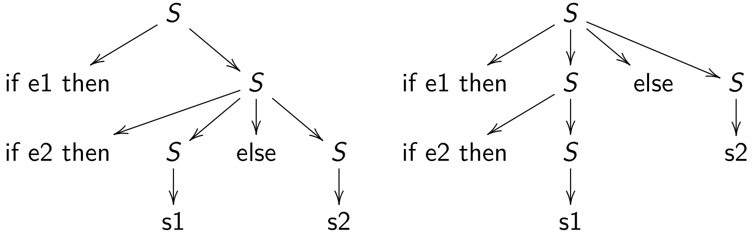
Example: If-Then and If-Then-Else

Consider the CFG

$$S \quad \rightarrow \quad \text{if } \textbf{bexp} \text{ then } S \mid \text{if } \textbf{bexp} \text{ then } S \text{ else } S \mid \textbf{prog}$$

where **bexp** and **prog** stand for boolean expressions and if statement-free programs respectively, defined elsewhere.

The string if e1 then if e2 then s1 else s2 then has two parse trees:



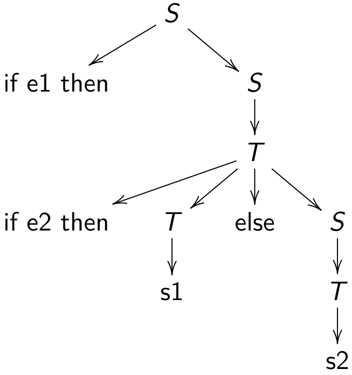
Example: If-Then and If-Then-Else

That grammar was **ambiguous**. But here's a grammar accepting the *exact same language* that is **unambiguous**:

$$S \rightarrow \text{ if } \textbf{bexp} \text{ then } S \mid T$$
$$T \rightarrow \text{ if } \textbf{bexp} \text{ then } T \text{ else } S \mid \textbf{prog}$$

There is now **only one** parse tree for if e1 then if e2 then s1 else s2.
(Given on the next slide.)

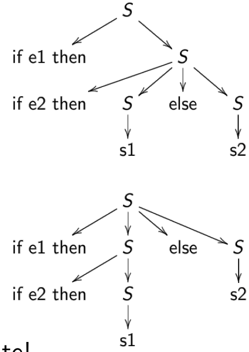
Example: If-Then and If-Then-Else



You **cannot** parse this string as if “e1 then (if e2 then s1) else s2”.

Q. Does that mean we can’t generate the ‘meaning’ of:
“e1 then (if e2 then s1) else s2”?

Reflecting on This Example



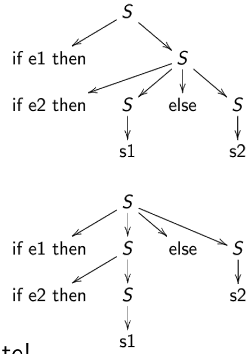
Observation.

- there's more than one grammar for a language
- some are ambiguous, others are not
- ambiguity is a property of *grammars*

Grammars for Programs.

- ambiguity is bad: don't know how program will execute!

Reflecting on This Example



Observation.

- there's more than one grammar for a language
- some are ambiguous, others are not
- ambiguity is a property of *grammars*

Grammars for Programs.

- ambiguity is bad: don't know how program will execute!
- replace ambiguous grammar with unambiguous one

Choices for converting ambiguous grammars to unambiguous ones

- *decide* on just *one* parse tree
- e.g. `if e1 then ( if e2 then s1 ) else s2` vs.
  `if e1 then ( if e2 then s1 else s2 )`
- in example: we have *chosen*:
  `if e1 then ( if e2 then s1 ) else s2`

What Ambiguity Isn't

Q. Is the grammar with the following production ambiguous?

$$T \quad \rightarrow \quad \text{if } \textbf{bexp} \text{ then } T \text{ else } S$$

Reasoning.

- Suppose that the above production was used
- we can then expand either T or S first.

What Ambiguity Isn't

Q. Is the grammar with the following production ambiguous?

$$T \quad \rightarrow \quad \text{if } \mathbf{bexp} \text{ then } T \text{ else } S$$

Reasoning.

- Suppose that the above production was used
- we can then expand either T or S first.

A. This is *not* ambiguity.

- both options give rise to the *same* parse tree
- indeed, for context-free languages it *doesn't* matter what production is applied first.
- thinking about parse trees, both expansions happen in parallel.

Main Message. Parse trees provide a better representation of syntax than derivations.

Inherently Ambiguous Languages

Q1. Can we always remove ambiguity?

Example. Language $L = \{a^i b^j c^k \mid (j = i \text{ or } j = k) \text{ and } i, j, k \in \mathbb{N}\}$

Q2. Why is this context-free?

Inherently Ambiguous Languages

Q1. Can we always remove ambiguity?

Example. Language $L = \{a^i b^j c^k \mid (j = i \text{ or } j = k) \text{ and } i, j, k \in \mathbb{N}\}$

Q2. Why is this context-free?

A. Note that $L = \{a^i b^i c^k \mid i, k \in \mathbb{N}\} \cup \{a^i b^j c^j \mid i, j \in \mathbb{N}\}$

- idea: start with production that "splits" between the union
- $S \to T \mid W$ where T is "left" and W is "right"

Complete Grammar. It starts with $S \ \to \ T \ \mid \ W$. Assume:

- left part uses non-terminals T, U, V
- right part uses non-terminals W, X, Y

Inherently Ambiguous Languages

Q1. Can we always remove ambiguity?

Example. Language $L = \{a^i b^j c^k \mid (j = i \text{ or } j = k) \text{ and } i, j, k \in \mathbb{N}\}$

Q2. Why is this context-free?

A. Note that $L = \{a^i b^i c^k \mid i, k \in \mathbb{N}\} \cup \{a^i b^j c^j \mid i, j \in \mathbb{N}\}$

- idea: start with production that "splits" between the union
- $S \to T \mid W$ where T is "left" and W is "right"

Complete Grammar. It starts with $S \to T \mid W$. Assume:

- left part uses non-terminals T, U, V
- right part uses non-terminals W, X, Y

$$\begin{array}{lll} T & \to & UV \\ U & \to & aUb \mid \epsilon \\ V & \to & cV \mid \epsilon \end{array} \qquad \begin{array}{lll} W & \to & XY \\ X & \to & aX \mid \epsilon \\ Y & \to & bYc \mid \epsilon \end{array}$$

Inherently Ambiguous Languages

Q1. Can we always remove ambiguity?

Example. Language $L = \{a^i b^j c^k \mid (j = i \text{ or } j = k) \text{ and } i, j, k \in \mathbb{N}\}$

Q2. Why is this context-free?

A. Note that $L = \{a^i b^i c^k \mid i, k \in \mathbb{N}\} \cup \{a^i b^j c^j \mid i, j \in \mathbb{N}\}$

- idea: start with production that "splits" between the union
- $S \to T \mid W$ where T is "left" and W is "right"

Complete Grammar. It starts with $S \;\to\; T \;\mid\; W$. Assume:

- left part uses non-terminals T, U, V
- right part uses non-terminals W, X, Y

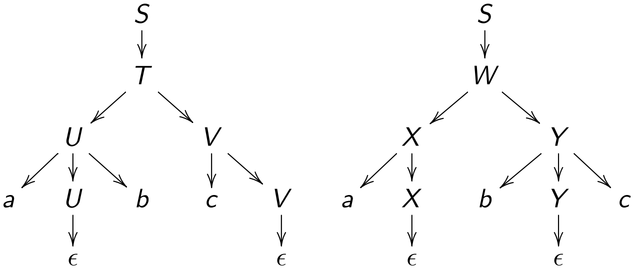
$$\begin{aligned} T &\to UV & \qquad W &\to XY \\ U &\to aUb \mid \epsilon & \qquad X &\to aX \mid \epsilon \\ V &\to cV \mid \epsilon & \qquad Y &\to bYc \mid \epsilon \end{aligned}$$

Q3. Why is this *language* ambiguous?

Inherently Ambiguous Languages

Problem. Both left part $a^i b^i c^k$ and right part $a^i b^j c^j$ has non-empty intersection: $a^i b^i c^i$

Ambiguity where a, b and c are equi-numerous, e.g., $a^1 b^1 c^1 = abc$:



So there are two parse trees for the same word!

Fact. There is *no* unambiguous grammar for this language (we don't prove this)

The Bad News

Q1. Can we *compute* an unambiguous grammar whenever one exists?

Q2. Can we even *determine* whether an unambiguous grammar exists?

A. If we interpret "compute" and "determine" as "by means of a program" (that works for an arbitrary CFL), then no.

- There is *no* program that solves this problem for *all* grammars
- input: CFG G, output: ambiguous or not. This problem is *undecidable*

(More undecidable problems next week!)

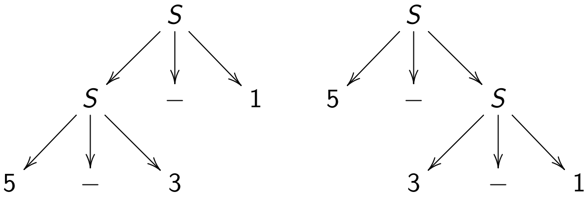
Example: Subtraction

Example.

$$S \quad \to \quad S - S \mid \mathrm{int}$$

- int stands for integers
- the intended meaning of $-$ is subtraction

Ambiguity.



Evaluation.

- left parse tree evaluates to 1
- right parse tree evaluates to 3
- so ambiguity matters! (As we also saw for the if/else statements.)

Technique 1: Associativity

Idea for ambiguity induced by binary operator (think: $-$)

- prescribe "implicit parentheses", e.g. $a - b - c \equiv (a - b) - c$
- make operator associate to the left or the right

Left Associativity.

$$S \quad \rightarrow \quad S - \textbf{int} \mid \textbf{int}$$

Result.

- $5 - 3 - 1$ can only be read as $(5 - 3) - 1$
- this is *left associativity*

Right Associativity.

$$S \rightarrow \textbf{int} - S \mid \textbf{int}$$

Idea. Break the symmetry

- one side of operator forced to lower level
- here: force right hand side of i to lower level
- create example derivation trees for all three grammars to see why that helps

Example: Multiplication and Addition

Example. Grammar for addition and multiplication

$$S \quad \to \quad S * S \mid S + S \mid \textbf{int}$$

Ambiguity.

- $1 + 2 * 3$ can be read as $(1 + 2) * 3$ and $1 + (2 * 3)$ with different results
- also $1 + 2 + 3$ is ambiguous – but this doesn't matter here.

Take 1. The trick we have just seen

- strictly evaluate from left to right

Example: Multiplication and Addition

Example. Grammar for addition and multiplication

$$S \quad \rightarrow \quad S * S \mid S + S \mid \textbf{int}$$

Ambiguity.

- $1 + 2 * 3$ can be read as $(1 + 2) * 3$ and $1 + (2 * 3)$ with different results
- also $1 + 2 + 3$ is ambiguous – but this doesn't matter here.

Take 1. The trick we have just seen

- strictly evaluate from left to right
- but this gives $1 + 2 * 3 \equiv (1 + 2) * 3$, *not* intended!

Goal. Want $*$ to have *higher precedence* than $+$

Technique 2: Precedence

Example Grammar giving $*$ higher precedence:

$$\begin{aligned} S &\to \; S + T \;\mid\; T \\ T &\to \; T * \textbf{int} \;\mid\; \textbf{int} \end{aligned}$$

Given e.g. $1 + 2 * 3$ or $2 * 3 + 1$

- *forced* to expand $+$ first: otherwise only $*$
- so $+$ will be *last* operation evaluated

Example. Derivation of $1 + 2 * 3$ (which we want to interpret as $1 + (2 * 3)$)

- suppose we start with $S \Rightarrow T \Rightarrow T * \mathrm{int}$
- stuck, as cannot generate $1 + 2$ from T

Idea. Forcing operation with *higher* priority to *lower* level

- three levels: S, (highest), T (middle) and integers
- lowest-priority operation generated by highest-level non-terminal

Example: Basic Arithmetic

Repeated use of $+$ and $*$:

$$\begin{aligned} S &\rightarrow S + T \mid S - T \mid T \\ T &\rightarrow T * U \mid T/U \mid U \\ U &\rightarrow (S) \mid \textbf{int} \end{aligned}$$

Main Differences.

- have *parentheses* to break operator priorities, e.g. $(1 + 2) * 3$
- parentheses at *lowest* level, so *highest* priority
- lower-priority operator can be inside parentheses
- expressions of arbitrary complexity (no nesting in previous examples)

Example: Balanced Brackets

$$S \quad \rightarrow \quad \epsilon \mid (S) \mid SS$$

Ambiguity.

- associativity: create brackets from left or from right (as before).
- at least wo ways of generating ():
  - $S \Rightarrow SS \Rightarrow S \Rightarrow (S) \Rightarrow ()$ and
  - $S \Rightarrow (S) \Rightarrow ()$
- indeed, *any* expression has *infinitely many* parse trees

Reason. More than one way to derive ϵ.

Technique 3: Controlling ϵ

Alternative Grammar with only *one* way to derive ϵ:

$$\begin{aligned} S &\to \epsilon \mid T \\ T &\to TU \mid U \\ U &\to () \mid (T) \end{aligned}$$

- ϵ can only be derived from S
- all other derivations go through T
- here: combined with multiple level technique
- ambiguity with ϵ can be hard to miss!

Pushdown Automata

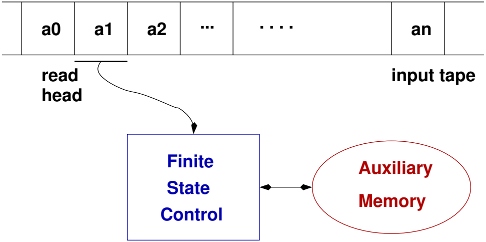
From Grammars to Automata

So Far.

- regular languages correspond to regular grammars (by definition).
- regular languages are exactly those accepted by FSAs or regular expressions (or regular grammars, of course).

Q. What automata correspond to *context-free* grammars?

General Structure of Automata



- *input tape* is a set of symbols
- *finite state control* is just like for DFAs / NFAs
- symbols are processed and head advances
- new aspect: *auxiliary memory*

Auxiliary Memory classifies languages and grammars

- no auxiliary memory: NFAs / DFAs: regular languages
- *stack*: *push-down automata*: context-free languages
- *unbounded tape*: Turing machines: all languages

PDAs cont'd

Actions of a push-down automaton

- change of internal state
- pushing or popping the stack
- advance to next input symbol

PDAs cont'd

Actions of a push-down automaton

- change of internal state
- pushing or popping the stack
- advance to next input symbol

Action dependencies. Actions generally depend on

- current state (of finite state control),
- input symbol, and
- symbol at the top of the stack

PDAs cont'd

Actions of a push-down automaton

- change of internal state
- pushing or popping the stack
- advance to next input symbol

Action dependencies. Actions generally depend on

- current state (of finite state control),
- input symbol, and
- symbol at the top of the stack

Acceptance. The machine accepts if

- input string is fully read
- machine is in accepting state

PDAs cont'd

Actions of a push-down automaton

- change of internal state
- pushing or popping the stack
- advance to next input symbol

Action dependencies. Actions generally depend on

- current state (of finite state control),
- input symbol, and
- symbol at the top of the stack

Acceptance. The machine accepts if

- input string is fully read
- machine is in accepting state

Variation.

- PDAs can *equivalently* be defined without final states F.
- Then, acceptance condition is having an empty stack (after the input word was completely read). **But we don't use that!**

Example

Language (that cannot be recognised by a DFA)

$$L = \{a^n b^n \mid n \geq 1\}$$

- *cannot* be recognised by a DFA
- *can* be generated by a context-free grammar
- *can* be recognised by a PDA

PDA design. (ad hoc, but showcases the idea)

- *phase 1:* (state s_1) *push* a's from the input onto the stack
- *phase 2:* (state s_2) *pop* a's from the stack, if there is a b on input
- *finalise:* if the input is exhausted and the stack is empty, enter a final state (s_3), i.e., accept the string.

Deterministic PDA – Definition

Definition. A *deterministic PDA* has the form $(S, s_0, F, \Sigma, \Gamma, Z, \delta)$, where

- S is the finite set of *states*, $s_0 \in S$ is the *initial state* and $F \subseteq S$ are the *final states*;
- Σ is the finite *alphabet*, or set of *input symbols*;
- Γ is the finite set of *stack symbols*, and $Z \in \Gamma$ is the *initial stack symbol*;
- δ is a (partial) *transition function*

$$\begin{array}{rcl} \delta \ : \ S \times (\Sigma \cup \{\epsilon\}) \times \Gamma & \nrightarrow & S \times \Gamma^* \\ \delta \ : \ (\text{state}, \text{input token or } \epsilon, \text{top-of-stack}) & \nrightarrow & (\text{new state}, \text{new top of stack}) \end{array}$$

Additional Requirement to ensure determinism:

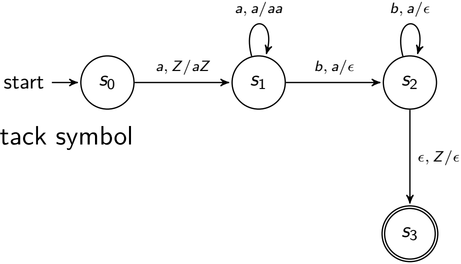
- if $\delta(s, \epsilon, \gamma)$ is defined, then $\delta(s, x, \gamma)$ is undefined for all $x \in \Sigma$ and $\gamma \in \Gamma$
- ensures that automaton has *at most* one execution

Notation

Given. Deterministic PDA with transition function

$$\delta \ : \ S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \quad \nrightarrow \quad S \times \Gamma^*$$

$$\delta \ : \ (\text{state}, \text{input token or } \epsilon, \text{top-of-stack}) \quad \nrightarrow \quad (\text{new state}, \text{new top of stack})$$



Notation.

- write $\delta(s, x, \gamma) = s'/\sigma$
- $\sigma \in \Gamma^*$ is a *string* that replaces top stack symbol
- *final states* are usually underlined ($\underline{s}$)

Rationale.

- *replacing* top stack symbol gives just *one* operation for push and pop
- pop: $\delta(s, x, \gamma) = s'/\epsilon$
- push: $\delta(s, x, \gamma) = s'/w\gamma$

Two types of PDA transition

Input-consuming transitions

- δ contains $(s_1, x, \gamma) \mapsto s_2/\sigma$
- automaton reads symbol x
- symbol x is consumed

Two types of PDA transition

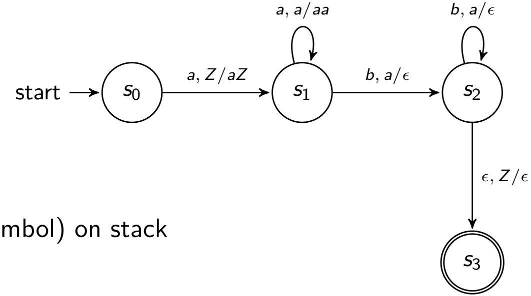
Input-consuming transitions

- δ contains $(s_1, x, \gamma) \mapsto s_2/\sigma$
- automaton reads symbol x
- symbol x is consumed

Non-consuming transitions

- independent of input symbol
- can happen *any time* and does not consume input symbol
- δ contains $(s_1, \epsilon, \gamma) \mapsto s_2/\sigma$
  Recall that for the pair s_1, γ, we can't have any other entry (s_1, x, γ) with $x \in \Sigma$ to stay deterministic! (See slide 50)
  Q. How is this different from epsilon transitions in ϵ-NFAs?

Example cont'd



Language $L = \{a^n b^n \mid n \geq 1\}$

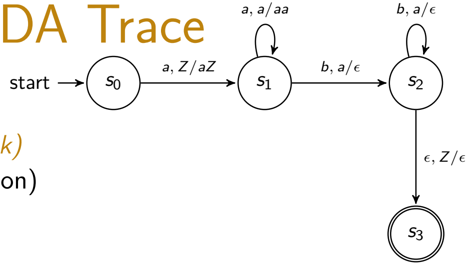
Push-down automaton

- starts with Z (initial stack symbol) on stack
- final state is s_3 (underlined)
- transition function (partial) given by

$$\begin{array}{lcll}
\delta(s_0, a, Z) & \mapsto & s_1/aZ & \text{push first } a \\
\delta(s_1, a, a) & \mapsto & s_1/aa & \text{push further } a\text{'s} \\
\delta(s_1, b, a) & \mapsto & s_2/\epsilon & \text{start popping } a\text{'s} \\
\delta(s_2, b, a) & \mapsto & s_2/\epsilon & \text{pop further } a\text{'s} \\
\delta(s_2, \epsilon, Z) & \mapsto & \underline{s_3}/\epsilon & \text{accept}
\end{array}$$

(δ is partial, i.e., undefined for many arguments)
Also note that we don't have to delete Z in the last step.
The stack doesn't have to be empty at the end.

Example cont'd — PDA Trace



PDA configurations

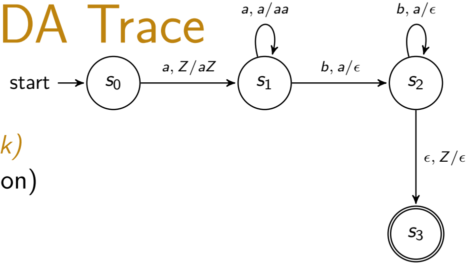
- triples: *(state, remaining input, stack)*
- top of stack on the *left* (by convention)

Example Execution.

$$(s_0, aaabbb, Z) \Rightarrow$$

(accept)

Example cont'd — PDA Trace



PDA configurations

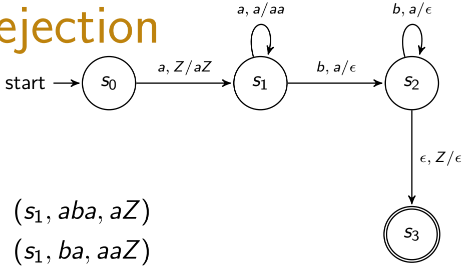
- triples: *(state, remaining input, stack)*
- top of stack on the *left* (by convention)

Example Execution.

$$
\begin{aligned}
(s_0, aaabbb, Z) &\Rightarrow (s_1, aabbb, aZ) && \text{(push first } a\text{)}\\
&\Rightarrow (s_1, abbb, aaZ) && \text{(push further } a\text{'s)}\\
&\Rightarrow (s_1, bbb, aaaZ) && \text{(push further } a\text{'s)}\\
&\Rightarrow (s_2, bb, aaZ) && \text{(start popping } a\text{'s)}\\
&\Rightarrow (s_2, b, aZ) && \text{(pop further } a\text{'s)}\\
&\Rightarrow (s_2, \epsilon, Z) && \text{(pop further } a\text{'s)}\\
&\Rightarrow (\underline{s_3}, \epsilon, \epsilon) && \text{(accept)}
\end{aligned}
$$

Accepting execution. Input exhausted, ends in final state (as usual!).

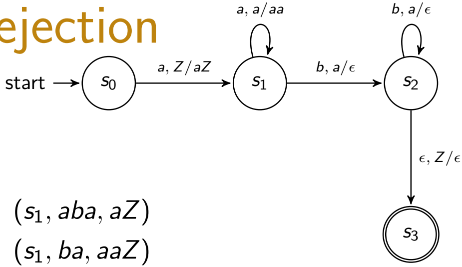
Example cont'd — Rejection



PDA execution.

$$\begin{aligned}(s_0, aaba, Z) &\Rightarrow (s_1, aba, aZ)\\ &\Rightarrow (s_1, ba, aaZ)\\ &\Rightarrow (s_2, a, aZ)\end{aligned}$$

Example cont'd — Rejection



PDA execution.

$$\begin{aligned}(s_0, aaba, Z) &\Rightarrow (s_1, aba, aZ)\\ &\Rightarrow (s_1, ba, aaZ)\\ &\Rightarrow (s_2, a, aZ)\\ &\Rightarrow \text{???}\end{aligned}$$

Non-accepting execution.

- No transition possible, stuck without reaching final state
- rejection happens when transition function is undefined for current configuration (state, input, top of stack) or when word is consumed, and no epsilon transitions can bring us to a final state.

Example: Palindromes with ‘Centre Mark’

Example Language.

$$L = \{wcw^R \mid w \in \{a,b\}^* \land w^R \text{ is } w \text{ reversed}\}$$

Deterministic PDA that accepts L

Example: Palindromes with 'Centre Mark'

Example Language.

$$L = \{wcw^R \mid w \in \{a, b\}^* \land w^R \text{ is } w \text{ reversed}\}$$

Deterministic PDA that accepts L

- Push a's and b's onto the stack as we seem them;
- When we see c, *change state*;
- Now try to match the tokens we are reading with the tokens on top of the stack, popping as we go;
- If the top of the stack is the empty stack symbol Z, enter the final state via an ϵ-transition. Hopefully our input has been used up too!

Exercise. Define this formally!

Non-Deterministic PDAs

Deterministic PDAs

- transitions are a partial function

$$\delta \; : \; S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \quad \nrightarrow \quad S \times \Gamma^*$$

$$\delta \; : \; (\text{state}, \text{input token or } \epsilon, \text{top-of-stack}) \quad \nrightarrow \quad (\text{new state}, \text{new top of stack})$$

- side condition about ϵ-transitions

Non-Deterministic PDAs

- transitions given by *relation*

$$\delta \; \subseteq \; S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \quad \times \quad S \times \Gamma^*$$

- no side condition (at all).

Main differences

- for deterministic PDA: *at most* one transition possible
- for non-deterministic PDA: *zero or more* transitions possible

Non-Deterministic PDAs cont'd

Finite Automata

- non-determinism is *convenient*
- but doesn't give extra power (subset construction)
- can convert every NFA to an equivalent DFA

Push-down Automata.

- non-determinism *gives* extra power
- cannot convert *every* non-deterministic PDA to deterministic PDA
- there are context- free languages that can *only* be recognised by non-deterministic PDA
- intuition: non-determinism allows "guessing"

Grammar / Automata correspondence

- non-deterministic PDAs are more important
- they correspond to context-free languages

Example: Even-Length Palindromes

Palindromes of even length, *without* centre-marks

$$L = \{ww^R \mid w \in \{a,b\}^* \wedge w^R \text{ is } w \text{ reversed}\}$$

- this is a context-free language
- *cannot* be recognised by deterministic PDA
- intuitive reason: no centre-mark, so don't know when first half of word is read

Non-deterministic PDA for L has the transition

$$\delta(s, \epsilon, \gamma) \quad = \quad r/x$$

- $x \in \{a, b, Z\}$, s is the 'push' state and r the 'match and pop' state.

Intuition

- "guess" (non-deterministically) whether we need to enter "match-and-pop" state
- automaton gets stuck if guess is not correct (no harm done)
- automaton accepts if guess is correct

Grammars and PDAs

Theorem. Context-free languages and *non-deterministic* PDAs are equivalent

- for every CFL *L* there exists a PDA that accepts *L*
- if *L* is accepted by a non-deterministic PDA, then *L* is a CFL.

Proof. We only do one direction: construct PDA from CFL (i.e., CFL to PDA).

- other direction (i.e., PDA to CFL) quite complex.
- for our proof, since we have a CFL, by definition, there is CFG.

From CFG to PDA

Given. Context-Free Grammar $G = (V_t, V_n, S, P)$

Construct non-deterministic PDA $A = (Q, q_0, F, \Sigma, \Gamma, Z, \delta)$

States. q_0 (initial state), q_1 (working state) and $\underline{q_2}$ (final state).

Alphabet. $\Sigma = V_t$, terminal symbols of the grammar

Stack Alphabet. $\Gamma = V_t \cup V_n \cup \{Z\}$

Initialisation.

- push start symbol S onto stack, enter working state q_1
- $\delta(q_0, \epsilon, Z) \mapsto q_1/SZ$

Termination.

- if the stack is empty (i.e., just contains Z), terminate
- $\delta(q_1, \epsilon, Z) \mapsto \underline{q_2}/\epsilon$

From CFGs to PDAs: working state

Idea.

- build the derivation on the stack by expanding non-terminals according to productions
- if a terminal appears that matches the input, pop it
- terminate, if the entire input has been consumed

Expand Non-Terminals.

- non-terminals on the stack are replaced by right hand side of productions
- $\delta(q_1, \epsilon, A) \mapsto q_1/\alpha$ for all productions $A \to \alpha$

Pop Terminals.

- terminals on the stack are popped if they match the input
- $\delta(q_1, x, x) \mapsto q_1/\epsilon$ for all terminals x

Result of Construction. *Non-deterministic* PDA

- may have more than one production for a non-terminal

Example — Derive a PDA for a CFG

Arithmetic Expressions as a grammar:

$$\begin{array}{rcl} S & \to & S + T \mid T \\ T & \to & T * U \mid U \\ U & \to & (S) \mid \textbf{int} \end{array}$$

1. Initialise:

$$\delta(q_0, \epsilon, Z) \mapsto q_1/SZ$$

2. Expand non-terminals:

$$\begin{array}{rcl} \delta(q_1, \epsilon, S) & \mapsto & q_1/S + T \\ \delta(q_1, \epsilon, S) & \mapsto & q_1/T \\ \delta(q_1, \epsilon, T) & \mapsto & q_1/T * U \end{array} \qquad \begin{array}{rcl} \delta(q_1, \epsilon, T) & \mapsto & q_1/U \\ \delta(q_1, \epsilon, U) & \mapsto & q_1/(S) \\ \delta(q_1, \epsilon, U) & \mapsto & q_1/\textbf{int} \end{array}$$

CFG to PDA cont'd

3. Match and pop terminals:

$$\begin{aligned}
\delta(q_1, +, +) &\mapsto q_1/\epsilon \\
\delta(q_1, *, *) &\mapsto q_1/\epsilon \\
\delta(q_1, \textbf{int}, \textbf{int}) &\mapsto q_1/\epsilon \\
\delta(q_1, (, () &\mapsto q_1/\epsilon \\
\delta(q_1,),)) &\mapsto q_1/\epsilon
\end{aligned}$$

4. Terminate:

$$\delta(q_1, \epsilon, Z) \mapsto \underline{q_2}/\epsilon$$

Example Trace

$$
\begin{aligned}
(q_0,\ \textbf{int} * \textbf{int},\ Z) &\Rightarrow (q_1,\ \textbf{int} * \textbf{int},\ SZ)\\
&\Rightarrow (q_1,\ \textbf{int} * \textbf{int},\ TZ)\\
&\Rightarrow (q_1,\ \textbf{int} * \textbf{int},\ T * UZ)\\
&\Rightarrow (q_1,\ \textbf{int} * \textbf{int},\ U * UZ)\\
&\Rightarrow (q_1,\ \textbf{int} * \textbf{int},\ \textbf{int} * UZ)\\
&\Rightarrow (q_1,\ *\textbf{int},\ *UZ)\\
&\Rightarrow (q_1,\ \textbf{int},\ UZ)\\
&\Rightarrow (q_1,\ \textbf{int},\ \textbf{int}Z)\\
&\Rightarrow (q_1,\ \epsilon,\ Z)\\
&\Rightarrow (q_2,\ \epsilon,\ \epsilon)\\
&\Rightarrow \quad \textit{accept}
\end{aligned}
$$

Summary about PDAs

- Definition of deterministic PDA
- Definition of non-deterministic PDA
- PDA configurations
- Relation of PDAs to CFGs/CFLs (same!)
- Compilation: CFGs to PDAs