# Formal Languages

## Parsing
## ISCL-BA-06

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2020/21

version: 86ec85d @2020-11-30

# What is a language?

- English, German, Chinese

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons
- Chemical formulas

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons
- Chemical formulas
- Arithmetic expressions

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons

- Chemical formulas
- Arithmetic expressions
- Python, Java, C++

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons

- Chemical formulas
- Arithmetic expressions
- Python, Java, C++
- XML, JSON, HTML, YAML

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons

- Chemical formulas
- Arithmetic expressions
- Python, Java, C++
- XML, JSON, HTML, YAML
- HTTP, TCP, UDP,

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons
- Chemical formulas
- Arithmetic expressions
- Python, Java, C++
- XML, JSON, HTML, YAML
- HTTP, TCP, UDP,
- The set of strings $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons
- Chemical formulas
- Arithmetic expressions
- Python, Java, C++
- XML, JSON, HTML, YAML
- HTTP, TCP, UDP,
- The set of strings $\{ba, baa, baaa, baaaa, \ldots\}$

# What is a language?

- English, German, Chinese
- Latin, Coptic, Sanskrit, Sumerian
- Proto-Germanic, Proto-Uralic, Proto-Dravidian
- Sign languages
- Esperanto
- Traffic signs, computer icons, emoticons

- Chemical formulas
- Arithmetic expressions
- Python, Java, C++
- XML, JSON, HTML, YAML
- HTTP, TCP, UDP,
- The set of strings $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$

According to Jurafsky and Martin (2009), the last set of strings form the 'sheep language'.

# Natural, artificial, formal languages

- Some languages in our list are natural languages
- In contrast, some are designed, they are artificial
- Formal languages are those that we can study formally
  - analyze them in principled ways
  - (provably) answer some questions about these languages
- All languages in our list can be studied as formal languages (to some extent)

# Languages as sets of strings

We define a *formal language* as a set of finite-length string over an *alphabet*.

- The sheep language from the first slide was represented as a set: $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$

# Languages as sets of strings

We define a *formal language* as a set of finite-length string over an *alphabet*.

- The sheep language from the first slide was represented as a set: $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$

  The alphabet of a language is the set of "symbols" in the language, conventionally denoted as $\Sigma$.
- For the sheep language, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

# Languages as sets of strings

We define a *formal language* as a set of finite-length string over an *alphabet*.

- The sheep language from the first slide was represented as a set: $\{ba, baa, baaa, baaaa, \ldots\}$

The alphabet of a language is the set of “symbols” in the language, conventionally denoted as $\Sigma$.

- For the sheep language, $\Sigma = \{a, b\}$
- What is the alphabet for English syntax?

# Formal grammar

A formal *grammar* is a finite specification of a (formal) language.

- Since we consider languages as sets of strings, for a finite language, we can (conceivably) list all strings
- How to define an infinite language?
- Is the definition $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$ 'formal enough'?

# Formal grammar

A formal *grammar* is a finite specification of a (formal) language.

- Since we consider languages as sets of strings, for a finite language, we can (conceivably) list all strings
- How to define an infinite language?
- Is the definition $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$ 'formal enough'?
- Using regular expressions, we can define it as $\mathtt{baa}^*$
- But we will introduce a more general method for defining languages soon

# Formal grammar

A formal *grammar* is a finite specification of a (formal) language.

- Since we consider languages as sets of strings, for a finite language, we can (conceivably) list all strings
- How to define an infinite language?
- Is the definition $\{\mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, \ldots\}$ 'formal enough'?
- Using regular expressions, we can define it as $\mathtt{baa}^*$
- But we will introduce a more general method for defining languages soon
- Are natural languages infinite?

# Formal languages

## Some definitions

Alphabet is the set of 'atomic' symbols in the language

String is a sequence of symbols from the alphabet, For example, $101100$ is a string over alphabet $\Sigma = \{0, 1\}$

- Concatenation: if $x = 10$ and $y = 11000101$, their concatenation $xy = 1011000101$
- We represent the empty string with $\epsilon$ (some books use $\lambda$)
- The notation $x^*$ indicates zero or more concatenation of string $x$ with itself, e.g., $\epsilon, 01, 010101$ (the operation is called Kleene star)
- The notation $x^+$ is a shorthand for $xx^*$
- $x^n$ means exactly $n$ repetition of string $x$

$\Sigma^*$ is all possible strings that can be defined over alphabet $\Sigma$

Sentence of a language is a string that is in the language (confusingly the term *word* is also common)

# Operations on languages

Since we define languages as sets, all set operations are applicable to languages. If $L_1$ and $L_2$ are languages,

- Intersection: $L_1 \cap L_2$
- Union: $L_1 \cup L_2$
- Difference: $L_1 - L_2$
- Complement: $\Sigma^* - L_1$
- Concatenation: $L_1 L_2 = \{xy | x \in L_1 \text{and} y \in L_2\}$

# Three different views on formal languages

- In formal language theory, a language is studied for itself. Languages are simply set of strings, we do not attach 'meaning' to them. The questions of interests are abstract. For example, 'how to find the intersection of two languages for which we have grammars?'
- In computer science, we want to analyze the structure (of, e.g., a computer program) to get some information, or 'meaning'. The most common area is compiler construction, but almost any syntactic analysis task is supported by formal definitions of the respective languages.
- In (computational) linguistics, the aim is to analyze sentences (syntax), and associate them with their meanings (semantics). Formal languages provide a way to study a seemingly chaotic object, natural language, in a principled way.

# Grammars: how to describe a language?

- In daily use, a 'grammar' is a book, it defines a language in detail
- But we are interested in more *formal* grammars
- The challenge is describing a possibly infinite set with a finite specification
- We already see that it was possible (e.g., regular expressions)
- Another possible way would be writing a computer program that determines if the given string is in the language
- However, we want more general descriptions: grammars that can describe any 'describable' language in a concise and easy to study formalism

# Grammars: how to describe a language?

- In daily use, a ‘grammar’ is a book, it defines a language in detail
- But we are interested in more *formal* grammars
- The challenge is describing a possibly infinite set with a finite specification
- We already see that it was possible (e.g., regular expressions)
- Another possible way would be writing a computer program that determines if the given string is in the language
- However, we want more general descriptions: grammars that can describe any ‘describable’ language in a concise and easy to study formalism

Aside: can any language be described by a finite description?

# Phrase structure grammars

- A phrase structure grammar is a generative device
- If a given string can be generated by the grammar, the string is in the language
- The grammar generates *all* and the *only* strings that are valid in the language
- A phrase structure grammar has the following components
  - $\Sigma$ A set of *terminal* symbols
  - $N$ A set of *non-terminal* symbols
  - $S \in N$ A special non-terminal, called the start symbol
  - $R$ A set of *rewrite rules* or *production rules* of the form:

$$\alpha \rightarrow \beta$$

which means that the sequence $\alpha$ can be rewritten as $\beta$ (both $\alpha$ and $\beta$ are sequences of terminal and non-terminal symbols)

# Phrase structure grammars

## Some conventions

- We use uppercase letters (sometimes capitalized words) for non-terminal symbols: A, B, C, NP, End
- We use lowercase letters (sometimes lowercase words) for terminals: a, b, c, cat, dog
- We use Greek letters letters for *sentential forms*, (sequences of terminal and non-terminal symbols): $\alpha$, $\beta$, $\gamma$
- For sequences of terminal symbols (strings) we use lowercase letters from the end of the alphabet: u, v, w, x, y, z

# Generating sentences from a PSG

1. Start with the symbol `S` as the first sentential form
2. Pick a rule with matching the part of the current sentential form
3. Apply the rewrite (production) rule
4. Repeat 2 and 3, until there are no non-terminals left

- Exhaustively exploring all possible productions ‘enumerates’ all sentences of the language described by the grammar

# Phrase structure grammars

## A very simple example – the sheep language

### A grammar

1. S $\rightarrow$ B A
2. B $\rightarrow$ b
3. A $\rightarrow$ a A
4. A $\rightarrow$ a

### An example derivation

| Sentential form | rule | notes |
|---|---|---|
| `S` | | start symbol |
| `BA` | S $\rightarrow$ B A | rule 1 |
| `bA` | B $\rightarrow$ b | rule 2 |
| `baA` | A $\rightarrow$ a A | rule 3 |
| `baaA` | A $\rightarrow$ a A | rule 3 |
| `baaa` | A $\rightarrow$ a | rule 4 |

# Phrase structure grammars

## A very simple example – the sheep language

### A grammar

1. $S \rightarrow B\ A$
2. $B \rightarrow b$
3. $A \rightarrow a\ A$
4. $A \rightarrow a$

Quick exercise: try to define a different grammar for the same language.

### An example derivation

| Sentential form | rule | notes |
|---|---|---|
| `S` | | start symbol |
| `BA` | $S \rightarrow B\ A$ | rule 1 |
| `bA` | $B \rightarrow b$ | rule 2 |
| `baA` | $A \rightarrow a\ A$ | rule 3 |
| `baaA` | $A \rightarrow a\ A$ | rule 3 |
| `baaa` | $A \rightarrow a$ | rule 4 |

# Generation to parsing

- The above procedure (generating all sentences from a generative grammar) gives us a possible way to do parsing:
  - Enumerate all sentences from the grammar
  - If the string we are interested comes out, it is in the language: parsing is successful
  - If it does not come out, it is not in the language: parsing failed (we'll get back to this point soon)
- We will also see later that this is in fact the idea behind top-down parsers

# Phrase structure grammars

## Another example: the goat language (a dialect of sheep language)[1]

**The grammar**

1. S → Begin B A End
2. B → b
3. A → a
4. A → a A
5. a A End → a ’ a
6. Begin b a → Begin b b a
7. Begin b b → b b

A few exercises:

- Describe the language
- Derive the string `bbaaa'a`
- Is the string `baa'a` in the language?
- Can you write a simpler grammar for this language?

[1] Some claim that the grammar is just the same, but goats use the word `m` instead of the word `b`.

# Phrase structure grammars

## A few notes

- The phrase structure grammars are not the only method for defining languages (sets)
- However, all known methods are either equivalent to, or less powerful than phrase structure grammars
- The formalism we sketched is general: any set (language) that can be generated by a computer program can be defined by a phrase structure grammar

# Languages and Grammars

## more definitions

- The language that can be derived from a grammar $G$, is denoted by $L(G)$
- The notation $u \Rightarrow v$ is used to denote 'immediate derivation', e.g., $A \Rightarrow aA$
- If a sentential form $\beta$ can be derived from another sentential form $\alpha$ with zero or more immediate derivations, we write $\alpha \overset{*}{\Rightarrow} \beta$
- I $\beta$ can be derived from $\alpha$ with exactly $n$ immediate derivations, we write $\alpha \overset{n}{\Rightarrow} \beta$
- Formally, $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$
- Two grammars $G$ and $G'$ are weakly equivalent if $L(G) = L(G')$

# The Chomsky hierarchy of grammars

| | |
|---|---|
| Type 0 | Unrestricted phrase structure grammars |
| Type 1 | Context-sensitive or monotonic grammars |
| Type 1.9 | Mildly-context sensitive grammars |
| Type 2 | Context-free grammars |
| Type 2.5 | Linear grammars |
| Type 3 | Regular grammars |
| Type 4 | Finite (choice) grammars |

# Type 0: unrestricted PSG

- As the names says - unrestricted, any form of the rewrite rules are allowed
- If a language can be generated at all, it can be defined/generated by a unrestricted PSG
- No general parsing algorithm exists, and in fact cannot exist
- In general, type 0 grammars are not interesting for practical applications
- The class of languages described by type 0 grammars is called *recursively enumerable* languages

# Type 1: monotonic

- We introduce one restriction to PSG: the right hand side (RHS) of a rule cannot be shorter than the left hand side (LHS)
- The rule applications cannot 'shrink' the sentential forms
- For example, our 'goat language grammar' is not monotonic, because of the rule Begin b b $\rightarrow$ b b
- This also means no $\epsilon$-rules
- Sometimes the language with only the empty string is allowed as an exception

# Type 1: context sensitive

- A context-sensitive grammar rewrites only one of its non-terminal on the LHS.
- Our 'goat language grammar' is not context-sensitive, because of the rule
- a A End $\rightarrow$ a ' a
- Context-sensitive and monotonic grammars are equivalent
- Parsing is possible with Type 1 grammars, but inefficient
- In general, not much practical use

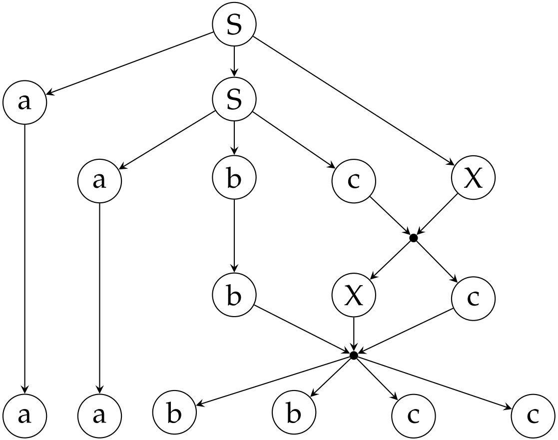
# An example type 1 grammar: $a^n b^n c^n$

$$
\begin{aligned}
S &\rightarrow abc \\
S &\rightarrow aSX \\
bXc &\rightarrow bbcc \\
cX &\rightarrow Xc
\end{aligned}
$$

# An example type 1 grammar: $a^n b^n c^n$

## monotonic version

$$
\begin{aligned}
S &\rightarrow abc \\
S &\rightarrow aSX \\
bXc &\rightarrow bbcc \\
cX &\rightarrow Xc
\end{aligned}
$$



# An example type 1 grammar: $a^n b^n c^n$

## context-sensitive version

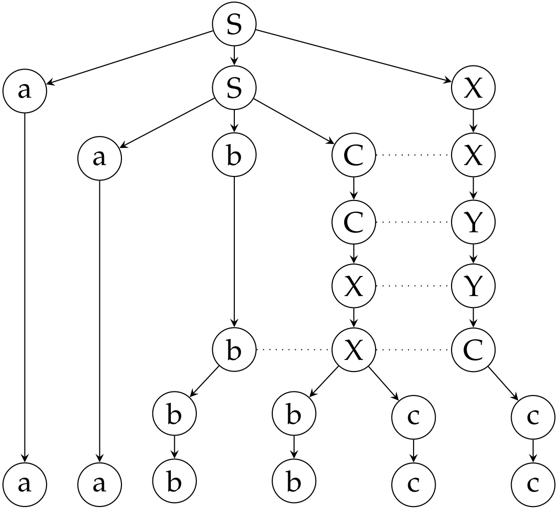
$$
\begin{aligned}
S &\rightarrow abC \\
S &\rightarrow aSX \\
bXC &\rightarrow bbCC \\
CX &\rightarrow CY \\
CY &\rightarrow XY \\
XY &\rightarrow XC \\
C &\rightarrow c
\end{aligned}
$$

# An example type 1 grammar: $a^n b^n c^n$

## context-sensitive version

$$
\begin{aligned}
S &\rightarrow abC \\
S &\rightarrow aSX \\
bXC &\rightarrow bbCC \\
CX &\rightarrow CY \\
CY &\rightarrow XY \\
XY &\rightarrow XC \\
C &\rightarrow c
\end{aligned}
$$

Exercise: try to write a (type 1) grammar for $a^n b^m c^n d^m$.



# Type 2: context free

- A context free language requires its LHS to have only a single non-terminal symbol. Rules are in the form

$$A \rightarrow \alpha$$

- This means the rewrite rules cannot be conditioned on context, they are independent of their environment
- It also means, each non-terminal defines its own language
- Context-free languages have efficient parsers, and used in practical applications
- All programming languages are (subclasses) of context free languages
- Most of natural language parsing is based on context-free parsing (more on this soon)

# Type 2: context free

## an example

$$
\begin{aligned}
\mathrm{Exp} &\rightarrow \mathrm{n} \\
\mathrm{Exp} &\rightarrow \mathrm{Exp\ Op\ Exp} \\
\mathrm{Exp} &\rightarrow (\ \mathrm{Exp}\ ) \\
\mathrm{Op} &\rightarrow + \\
\mathrm{Op} &\rightarrow - \\
\mathrm{Op} &\rightarrow \times \\
\mathrm{Op} &\rightarrow /
\end{aligned}
$$

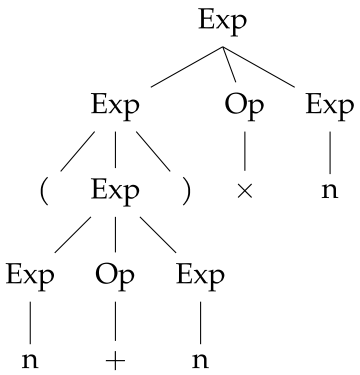
Generating $(n + n) \times n$

# Type 2: context free

## an example

$$
\begin{aligned}
\text{Exp} &\rightarrow \text{n} \\
\text{Exp} &\rightarrow \text{Exp Op Exp} \\
\text{Exp} &\rightarrow (\ \text{Exp}\ ) \\
\text{Op} &\rightarrow + \\
\text{Op} &\rightarrow - \\
\text{Op} &\rightarrow \times \\
\text{Op} &\rightarrow /
\end{aligned}
$$

Generating $(n + n) \times n$



# Recursion

- The notion of recursion is important grammars
- A CF rule is *directly recursive*, if RHS includes the non-terminal on the LHS symbol

$$\begin{aligned} \mathrm{A} &\rightarrow \mathrm{A}\ \alpha && \text{left recursive} \\ \mathrm{A} &\rightarrow \alpha\ \mathrm{A} && \text{right recursive} \\ \mathrm{A} &\rightarrow \alpha\ \mathrm{A}\ \beta && \text{self embedding} \end{aligned}$$

- Recursion can also be indirect:
  $\mathrm{A} \rightarrow \mathrm{B\ c\ B} \rightarrow \mathrm{d\ A}$
- Note that CF grammars are monotonic, unless they have $\epsilon$ rules

# CF grammars: notational variants

**Backus-Naur form (BNF)**

$$
\begin{aligned}
\text{Exp} &::= \text{n} \\
\text{Exp} &::= \langle\text{Exp}\rangle\ \langle\text{Op}\rangle\ \langle\text{Exp}\rangle \\
\text{Exp} &::= (\ \langle\text{Exp}\rangle\ ) \\
\text{Op} &::= + \\
\text{Op} &::= - \\
\text{Op} &::= \times \\
\text{Op} &::= /
\end{aligned}
$$

- Common in compiler generators and similar tools
- Also common standard definitions (e.g., HTML, XML)
- Non-terminals are put in angle brackets
- Instead of $\rightarrow$, we have ::=
- There are extended forms (EBNF, or extended CFG), e.g., allowing regexp

# Type 3: regular

- Regular grammars come in two flavors: *right-regular* and *left-regular*
- A right-regular grammar allows only two types of rules:
  $A \rightarrow a$ and $A \rightarrow a\,B$
- A left-regular grammar allows:
  $A \rightarrow a$ and $A \rightarrow B\,a$
- Generally, $\epsilon$-rules are also allowed $A \rightarrow \epsilon$
- Right-regular grammars are more common in practical use
- Almost all operations on regular languages are efficient, lots of practical use
- Regular grammars are equivalent to regular expressions

# Type 3: regular

## an example (right regular)

Generating 'baaa'

**Sheep language**

$$\begin{aligned} S &\rightarrow b\,A \\ A &\rightarrow a \\ A &\rightarrow a\,A \end{aligned}$$

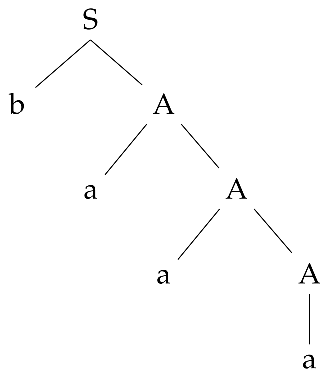
# Type 3: regular

## an example (right regular)

**Sheep language**

$$S \rightarrow b\ A$$
$$A \rightarrow a$$
$$A \rightarrow a\ A$$

Generating ‘baaa’



# Type 3: regular

## an example (left regular)

Generating 'baaa'

**Sheep language**

$$
\begin{aligned}
S &\rightarrow B\ a \\
B &\rightarrow b \\
B &\rightarrow B\ a
\end{aligned}
$$

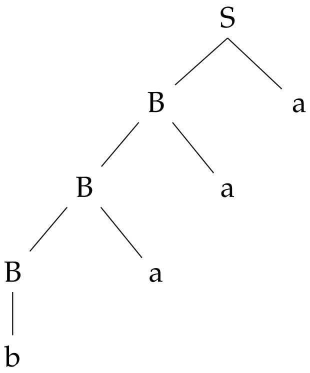
# Type 3: regular

## an example (left regular)

**Sheep language**

$$\begin{aligned} S &\rightarrow B\,a \\ B &\rightarrow b \\ B &\rightarrow B\,a \end{aligned}$$

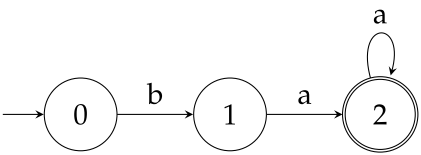
Generating 'baaa'



# Regular grammars, regular expressions, and finite-state automata

**Sheep language**

$$
\begin{aligned}
S &\rightarrow b\,A \\
A &\rightarrow a \\
A &\rightarrow a\,A
\end{aligned}
$$




$\mathtt{baa}^*$

# Chomsky hierarchy

## a summary and relation to automata

| Grammar | Language | Automata |
|---|---|---|
| Type 0 (unrestricted) | Recursively enumerable | Turing machines |
| Type 1 (context-sensitive) | Context sensitive | Linear bounded automata |
| Type 2 (context-free) | Context fee | Pushdown automata |
| Type 3 (regular) | Regular | Finite-state automata |

- Other theoretically (or practically) interesting classes exist
- Our focus in this course will be mainly context-free grammars
- A question: what does it mean for a grammar to be more expressive?

# Actually enumerating all sentences from a grammar

- As we sketched it earlier:
  1. Start with sentential form 'S'
  2. Pick a LHS that matches part of the sentential form
  3. Rewrite the part of the sentential form
  4. Repeat 2 & 3 until either
     - no non-terminals left in the sentential form: result is a sentence
     - there are no possible productions: dead end
- So far, we picked the rules manually, two strategies to do this automatically:
  - Explore all possible productions simultaneously
  - Use recursion or (iteration with an 'agenda'), and backtrack when we hit a dead end (or generated a sentence successfully)

# Example generation

S

**Another grammar for the sheep language**

$$
\begin{aligned}
S &\rightarrow B\ A \\
A &\rightarrow a \\
A &\rightarrow a\ A \\
BA &\rightarrow b\ A
\end{aligned}
$$

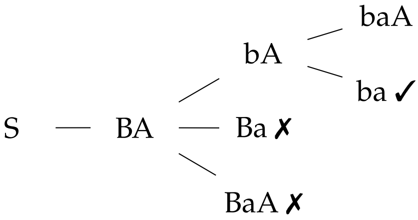
# Example generation

S — BA

**Another grammar for the sheep language**

$$
\begin{aligned}
S &\rightarrow B\ A \\
A &\rightarrow a \\
A &\rightarrow a\ A \\
BA &\rightarrow b\ A
\end{aligned}
$$

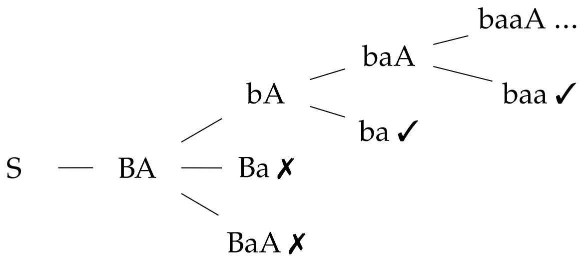
# Example generation




**Another grammar for the sheep language**

$$
\begin{aligned}
S &\rightarrow B\ A\\
A &\rightarrow a\\
A &\rightarrow a\ A\\
BA &\rightarrow b\ A
\end{aligned}
$$

# Example generation




**Another grammar for the sheep language**

$$
\begin{aligned}
S &\rightarrow B\ A \\
A &\rightarrow a \\
A &\rightarrow a\ A \\
BA &\rightarrow b\ A
\end{aligned}
$$

Note that we need to explore all options type 0 and type 1 grammars.

# Generation and parsing

## why unrestricted grammars are undecidable

- The generation procedure we outline can generate all sentence from any PSG
- We can define parsing as waiting until the string we want to parse comes out
- For monotonic/context-sensitive grammars, we can ensure to enumerate shortest strings first
- For unrestricted grammars, the sentential forms may shrink, as a result
  - if the string comes out, parsing is successful
  - if not, we do not know if it is not in the language, or we haven't obtained it yet

# How do we know a language is regular?

- Easy ways of proving that a language is regular: find one of
  - type 3 grammar
  - regular expression
  - finite-state automata

  that generates and recognizes the language

# How do we know a language is regular?

## the goat language

**regex examples**

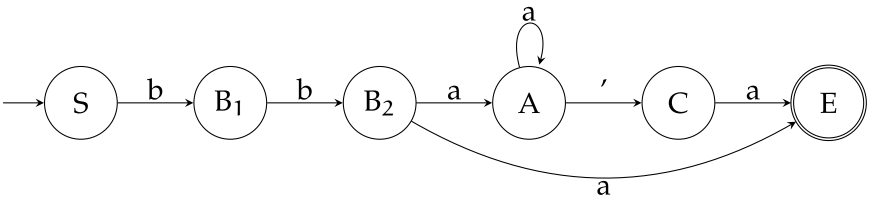
$\mathtt{bb(a \mid a*'a)}$

$\mathtt{bba(a*'a)?}$

**regular grammar**

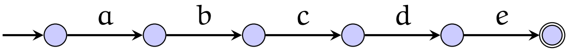
| | | | |
|---|---|---|---|
| $S \to bB$ | $B \to a$ | $A \to aA$ | $C \to {'}E$ |
| $B \to bB$ | $B \to aA$ | $A \to aC$ | $E \to a$ |

**finite-state automaton**



# How do we know a language is *not* regular?

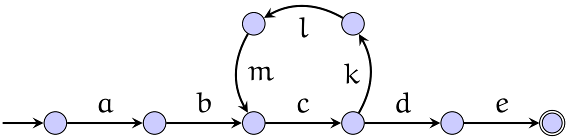
## pumping lemma for regular languages




- What is the length of longest string generated by this FSA?

# How do we know a language is *not* regular?

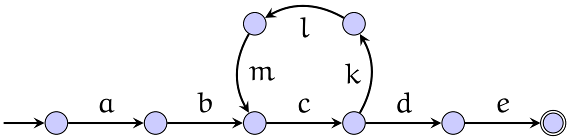
## pumping lemma for regular languages




- What is the length of longest string generated by this FSA?

# How do we know a language is *not* regular?

## pumping lemma for regular languages




- What is the length of longest string generated by this FSA?
- Any FSA generating an infinite language has to have a loop (application of recursive rule(s) in the grammar)
- Part of every string longer than some number will include repetition of the same substring (‘cklm’ above)

# Pumping lemma

## definition

For every regular language $L$, there exist an integer $p$ such that a string $x \in L$ can be factored as $x = uvw$,

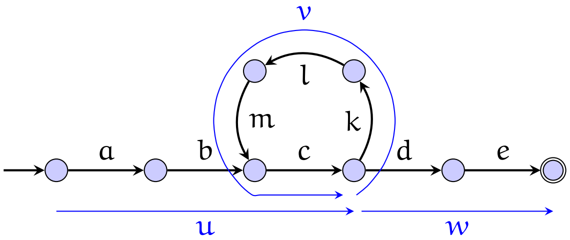
- $uv^i w \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$

# Pumping lemma

## definition

For every regular language $L$, there exist an integer $p$ such that a string $x \in L$ can be factored as $x = uvw$,

- $uv^iw \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$



# How to use pumping lemma

- We use pumping lemma to prove that a language is not regular
- Proof is by contradiction:
  - Assume the language is regular
  - Find a string $x$ in the language, for all splits of $x = uvw$, at least one of the pumping lemma conditions does not hold
    - $uv^iw \in L \ (\forall i \geqslant 0)$
    - $v \neq \epsilon$
    - $|uv| \leqslant p$

# Pumping lemma example

## prove $L = a^n b^n$ is not regular

- Assume $L$ is regular: there must be a $p$ such that, if $uvw$ is in the language
  1. $uv^iw \in L$ $(\forall i \geqslant 0)$
  2. $v \neq \epsilon$
  3. $|uv| \leqslant p$
- Pick the string $a^p b^p$
- For the sake of example, assume $p = 5$, $x = aaaaabbbbb$
- Three different ways to split

| | |
|---|---|
| $\underbrace{a}_{u}\underbrace{aaa}_{v}\underbrace{abbbbb}_{w}$ | violates 1 |
| $\underbrace{aaaa}_{u}\underbrace{ab}_{v}\underbrace{bbbb}_{w}$ | violates 1 & 3 |
| $\underbrace{aaaaab}_{u}\underbrace{bbb}_{v}\underbrace{b}_{w}$ | violates 1 & 3 |

# How do we know a language is context-free?

- Again, find a context-free grammar that generates the language
- Examples: $a^n b^n$

# How do we know a language is context-free?

- Again, find a context-free grammar that generates the language
- Examples: $a^n b^n$

$$
\boxed{\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \epsilon \end{aligned}}
$$

This is for $n \geqslant 0$, to disallow allow $a^0 b^0$, replace the second rule with $S \rightarrow ab$

# How do we know a language is *not* context-free?

## pumping lemma for context-free languages

- The idea is similar to regular languages, but we can have 'embedded' structures as well as simple loops
- For any sufficiently long sentence $uvxyz$ in a context-free language
  1. $uv^ixy^iz \in L \ (\forall i \geqslant 0)$
  2. $|vy| \geqslant 0$
  3. $|vxy| \leqslant p$
- Again, the proof is by contradiction
  - Assume the language is context-free
  - Find a string $s = uvxyz$ and a number $p$ in the language that does not satisfy the conditions above

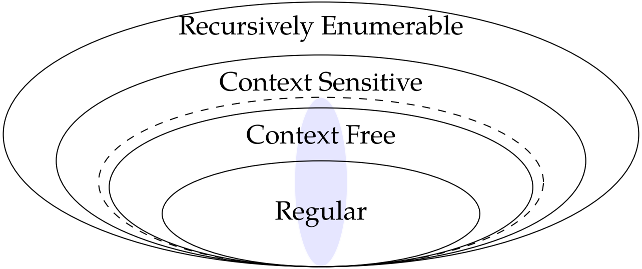
# Where do natural language syntax fit?

## Cross-serial dependencies

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Jan | säit | das | mer | em Hans | es huss | hälfed | aastriiche |
| Jan | said | that | we | Hans (DAT) | the house (ACC) | helped | paint |

- The above structure is not possible to parse using context-free languages
- Otherwise, experience so far indicates that a CF-based grammar can describe natural language syntax

# Chomsky hierarchy: the picture




- Chomsky hierarchy of languages form a hierarchy (with some care about empty language)
- It is often claimed that mildly context sensitive grammars (dashed ellipse) are adequate for representing natural languages
- Note, however, not even every regular language is a potential natural language (e.g., $\mathtt{a}^*\mathtt{bbc}^*$). The possible natural languages probably cross-cut this hierarchy (shaded region)

# Summary

- Phrase structure grammars are generative grammars that are finite specifications of (infinite) languages
- They form the basis of the theory of parsing
- More expressive grammar classes (type 0 and type 1) are not computationally attractive
- We will focus on more practical grammar classes, mainly context-free grammars, for the rest of the course

# Summary

- Phrase structure grammars are generative grammars that are finite specifications of (infinite) languages
- They form the basis of the theory of parsing
- More expressive grammar classes (type 0 and type 1) are not computationally attractive
- We will focus on more practical grammar classes, mainly context-free grammars, for the rest of the course
- Next: introduction to parsing
- Suggested reading: **grune2008**

# Example: deriving bbaaa'a

| Sentential form | rule |
|---|---|
| `S` | (init) |
| `Begin B A End` | S → Begin B A End |
| `Begin b A End` | B → b |
| `Begin b a A End` | A → a A |
| `Begin b a a A End` | A → a A |
| `Begin b a a A End` | A → a A |
| `Begin b a a a A End` | A → a A |
| `Begin b a a a a End` | A → a |
| `b b a a a a End` | Begin b a → b b a |
| `b b a a a ' a` | a a End → a ' a |

**The grammar**

1. S → Begin B A End
2. B → b
3. A → a
4. A → a A
5. a A End → a ' a
6. Begin b a → b b a
7. Begin b b → b b

# Example: deriving baa’a

| Sentential form | rule |
|---|---|
| `S` | (init) |
| `Begin B A End` | S $\rightarrow$ Begin B A End |
| `Begin b A End` | B $\rightarrow$ b |
| `Begin b a A End` | A $\rightarrow$ a A |
| `Begin b a a A End` | A $\rightarrow$ a A |
| `Begin b a a A End` | A $\rightarrow$ a A |
| `Begin b a a ' a` | (none) |

We are stuck with a sentential form with non-terminals.

**The grammar**

1. S $\rightarrow$ Begin B A End
2. B $\rightarrow$ b
3. A $\rightarrow$ a
4. A $\rightarrow$ a A
5. a A End $\rightarrow$ a ’ a
6. Begin b a $\rightarrow$ b b a
7. Begin b b $\rightarrow$ b b

### $a^n b^m c^n d^m$

1. $S \rightarrow X\,Y$
2. $X \rightarrow aXC$
3. $X \rightarrow aC$
4. $Y \rightarrow BYd$
5. $Y \rightarrow Bd$
6. $CB \rightarrow BC$
7. $aB \rightarrow ab$
8. $bB \rightarrow bb$
9. $Cd \rightarrow cd$
10. $Cc \rightarrow cc$

$a^n b^m c^n d^m$

1. $S \rightarrow X\,Y$
2. $X \rightarrow aXC$
3. $X \rightarrow aC$
4. $Y \rightarrow BYd$
5. $Y \rightarrow Bd$
6. $CB \rightarrow BC$
7. $aB \rightarrow ab$
8. $bB \rightarrow bb$
9. $Cd \rightarrow cd$
10. $Cc \rightarrow cc$

Some explanation:

- Rule (1) generates a string with two parts X and Y
- For X, we generate as many C's as a's (3), and for Y, we generate as many B's as d's (4)
- We will eventually rewrite Cs as c, and 'B's as 'b', but their order is not correct.
- When recursions for X and Y terminate, we have equal number of a's and C, and equal number of B's d's, and a's are all at the beginning, and d's are all at the end
- Rule (2) swaps B and C's
- We allow rewriting B as b only after and a or b, and allow rewriting C as C only before d

# Acknowledgments, references, additional reading material

- Please read **grune2008** chapter 2, a large part of the lecture follows this chapter