02141 Computer Science Modelling
Context-Free Languages

CFL2: Grammars and data, ambiguities, associativity and precedence

Last Time on Context-Free Languages...

A **context-free grammar** is a quadruple $G = (V, T, P, S)$ where:

- V is a finite set of symbols called the **variables** or **non-terminals**.
- T is a finite set of symbols, disjoint from V, called the **terminals**.
- $P \subseteq V \times (V \cup T)^\star$ is a finite set of **productions**.
  where a rule $(A, \alpha) \in P$ is written $A \to \alpha$.
- $S \in V$ is the **start symbol**.

For $A \to \alpha_1, \ldots, A \to \alpha_n$, we use the short-hand:

$$A \to \alpha_1 \mid \cdots \mid \alpha_n$$

Last Time on Context-Free Languages. . .

Several equivalent ways to infer strings from a grammar:

1. Derivation/top down: $S \overset{\star}{\Rightarrow} w$
   - applying rules "forward", from the starting symbol S
   - to eventually obtain a word w of terminal symbols
2. Recursive inference/bottom up:
   - Applying rules "backwards"
   - Inferring step by step which words must be derivable from each variable symbol.
3. Parse Trees:
   - depicting both the word (at the leaves)
   - and its derivation from the starting symbol (the root)

The **language** of a grammar $G = (V, T, P, S)$ is defined as:

$$L(G) = \{ w \in T^\star \mid S \overset{\star}{\Rightarrow} w \}$$

Grammars and Data

We live in a **connected world**, where our data is shared among many different distributed systems.

Grammars and Data

We live in a **connected world**, where our data is shared among many different distributed systems.

But it can be a nightmare trying to transfer data between **different applications**.

Grammars and Data

Context-free grammars can save the day!

HyperText Markup Language (HTML)

One of the most widely known formats for sharing data is the **HyperText Markup Language (HTML)**.

Example: Today teacher's schedule:

```
<h1>TODO List for Today</h1>
<p>Things to do <i>before lunch</i>:
<ol>
    <li>Review papers for conference.</li>
    <li>Meeting with head of department.</li>
    <li>Randomly fail a few students for fun.</li>
    <li>Continue work on research paper.</li>
</ol>
</p>
```

Data is surrounded by **tags**, which tell the web browser how to display it.

We can describe HTML using a **context-free grammar**!

HyperText Markup Language (HTML)

Let us consider a fragment of the **grammar for HTML**:

HyperText Markup Language (HTML)

Let us consider a fragment of the **grammar for HTML**:

A *Document* is a possibly empty sequence of *Elements*

$$\begin{array}{rcl} \mathit{Document} & \to & \epsilon \\ & \mid & \mathit{Element}\ \mathit{Document} \end{array}$$

An *Element* can be *Text* or a paragraph with a *Document* or a *List* or ...

$$\begin{array}{rcl} \mathit{Element} & \to & \mathit{Text} \\ & \mid & < \mathtt{p} > \mathit{Document} < /\mathtt{p} > \\ & \mid & < \mathtt{ol} > \mathit{List} < /\mathtt{ol} > \\ & \mid & \ldots \end{array}$$

A *List* is a possibly empty sequence of *Documents* enclosed by `li` tags.

$$\begin{array}{rcl} \mathit{List} & \to & \epsilon \\ & \mid & < \mathtt{li} > \mathit{Document} < /\mathtt{li} > \mathit{List} \end{array}$$

eXtensible Markup Language (XML)

The **eXtensible Markup Language (XML)** looks syntactically similar to HTML, but allows us to store more general **structured data**.

The main difference is that **we get to define our own tags**!

Example: XML for email messages

```
<email>
    <from>teacher@dtu.dk</from>
    <to>s007@dtu.dk</to>
    <to>s008@dtu.dk</to>
    <subject>Mandatory assignment</subject>
    <body>Hi, remember to submit your assignment today.</body>
</email>
```

eXtensible Markup Language (XML)

How do we specify the nesting of the tags?

eXtensible Markup Language (XML)

How do we specify the nesting of the tags? **With a grammar!**

eXtensible Markup Language (XML)

How do we specify the nesting of the tags? **With a grammar!**
We can write a **Document-Type Definition (DTD)** or an **XML Schema** that describes the grammar of the XML file

eXtensible Markup Language (XML)

How do we specify the nesting of the tags? **With a grammar!**
We can write a **Document-Type Definition (DTD)** or an **XML Schema** that describes the grammar of the XML file
Example: DTD for the previous XML file.

```
<!DOCTYPE email [
    <!ELEMENT email (from,to+,bcc*,subject,body)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT bcc (#PCDATA)>
    <!ELEMENT subject (#PCDATA)>
    <!ELEMENT body (#PCDATA)>
]>
```

eXtensible Markup Language (XML)

How do we specify the nesting of the tags? **With a grammar!**
We can write a **Document-Type Definition (DTD)** or an **XML Schema** that describes the grammar of the XML file
Example: DTD for the previous XML file.

```
<!DOCTYPE email [
    <!ELEMENT email (from,to+,bcc*,subject,body)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT bcc (#PCDATA)>
    <!ELEMENT subject (#PCDATA)>
    <!ELEMENT body (#PCDATA)>
]>
```

Each `<!ELEMENT...>` item defines a tag (a **variable**), and the content of the tag — a regular expression of other tags (the **body** of a production).

eXtensible Markup Language (XML)

How do we specify the nesting of the tags? **With a grammar!**
We can write a **Document-Type Definition (DTD)** or an **XML Schema** that describes the grammar of the XML file
Example: DTD for the previous XML file.

```
<!DOCTYPE email [
    <!ELEMENT email (from,to+,bcc*,subject,body)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT bcc (#PCDATA)>
    <!ELEMENT subject (#PCDATA)>
    <!ELEMENT body (#PCDATA)>
]>
```

Each `<!ELEMENT...>` item defines a tag (a **variable**), and the content of the tag — a regular expression of other tags (the **body** of a production). `#PCDATA` stands for **Parsed Character Data**, meaning arbitrary text that does not contain any tags.

Exercise 2.1

We can convert a DTD into a **context-free grammar**!

IDEA: use the translation we saw in the exercise session of lecture 1.

Translate the DTD for emails of the previous slide into a context-free grammar.

JavaScript Object Notation (JSON)

JSON values can be *String*s, *Number*s, *Obj*ects, *Array*s, or the keywords *true*, *false* and *null*:

$$Value \rightarrow String \mid Number \mid Obj \mid Array \mid true \mid false \mid null;$$

JSON *Obj*ects are possibly empty sequences of *Pair*s, where *Pair*s are separated by colons "," and the whole sequence is enclosed by curly brackets { , }:

$$Obj \rightarrow \dots$$

JSON *Pair*s are defined by a *String* and a *Value*, separated by ":".

$$Pair \rightarrow \dots$$

JSON *Array*s are possibly empty sequences of *Value*s. *Value*s are separated by "," and the whole sequence is enclosed with square brackets "[", "]":

$$Array \rightarrow \dots$$

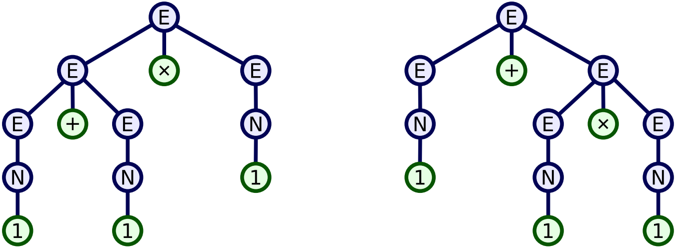
Exercise 2.2

Consider the incomplete JSON grammar of the previous slide and complete the productions of by filling the dots and adding additional non-terminals and their productions if necessary.

Last Time on Context-Free Languages...

$$\begin{array}{lclclclcl} E & \rightarrow & N & \mid & E+E & \mid & E \times E & \mid & (E) \\ N & \rightarrow & 0 & \mid & 1 & \mid & N0 & \mid & N1 \end{array}$$

There are multiple parse trees for $1 + 1 \times 1$:

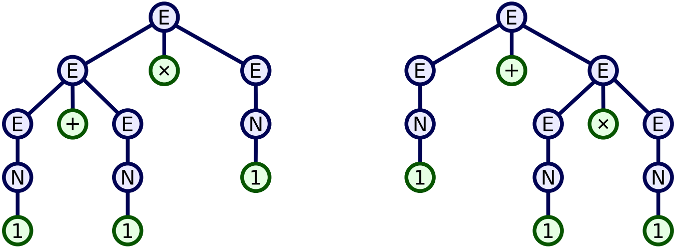


Another example: $1 + 1 + 1$

Last Time on Context-Free Languages. . .

$$\begin{array}{lclclclcl} E & \to & N & \mid & E + E & \mid & E \times E & \mid & (E) \\ N & \to & 0 & \mid & 1 & \mid & N0 & \mid & N1 \end{array}$$

There are multiple parse trees for $1 + 1 \times 1$:



Another example: $1 + 1 + 1$

We call a context-free grammar **ambiguous** if there exists a string for which we can build more than one **parse tree**.

Ambiguous Grammars

Where does the **ambiguity** come from?

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

We probably want the second interpretation: multiplication has a **higher precedence** (binds tighter) than addition.

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

We probably want the second interpretation: multiplication has a **higher precedence** (binds tighter) than addition.

Example: Associativity of operators: how do we read $2 - 1 - 1$?

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

We probably want the second interpretation: multiplication has a **higher precedence** (binds tighter) than addition.

Example: Associativity of operators: how do we read $2 - 1 - 1$?

- As $(2 - 1) - 1 = 0$?
- As $2 - (1 - 1) = 2$?

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

We probably want the second interpretation: multiplication has a **higher precedence** (binds tighter) than addition.

Example: Associativity of operators: how do we read $2 - 1 - 1$?

- As $(2 - 1) - 1 = 0$?
- As $2 - (1 - 1) = 2$?

Most operators are left-associative (first interpretation), but some are right-associative.

Ambiguous Grammars

Where does the **ambiguity** come from?

Example: Precedence of operators: how do we read $1 + 1 \times 0$?

- As $(1 + 1) \times 0 = 0$?
- As $1 + (1 \times 0) = 1$?

We probably want the second interpretation: multiplication has a **higher precedence** (binds tighter) than addition.

Example: Associativity of operators: how do we read $2 - 1 - 1$?

- As $(2 - 1) - 1 = 0$?
- As $2 - (1 - 1) = 2$?

Most operators are left-associative (first interpretation), but some are right-associative.

Modern parsers allow one to specify operator precedence and associativity but **can we fix this directly in the grammar?**

Enforcing associativity

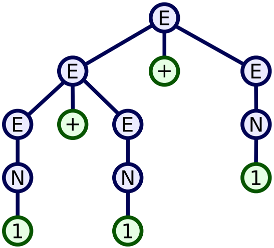
Let us start with **ambiguity** in this simple grammar

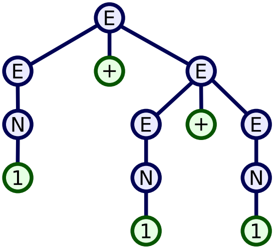
$$E \quad \rightarrow \quad E + E \quad | \quad N \quad | \quad (E)$$

Enforcing associativity

Let us start with **ambiguity** in this simple grammar

$$E \quad \rightarrow \quad E + E \quad | \quad N \quad | \quad (E)$$





Both parse trees yield the string $1 + 1 + 1$.

Enforcing associativity

But $+$ is associative, e.g.

- As $(1+1)+1 = 3$?
- As $1+(1+1) = 3$?

so **who cares**, if it gives the same result?

Enforcing associativity

But $+$ is associative, e.g.

- As $(1 + 1) + 1 = 3$?
- As $1 + (1 + 1) = 3$?

so **who cares**, if it gives the same result?

Not all operators are associative:

Enforcing associativity

But $+$ is associative, e.g.

- As $(1+1)+1=3$?
- As $1+(1+1)=3$?

so **who cares**, if it gives the same result?

Not all operators are associative:

- $(4-2)-2 \neq 4-(2-2)$

Enforcing associativity

But $+$ is associative, e.g.

- As $(1+1)+1=3$?
- As $1+(1+1)=3$?

so **who cares**, if it gives the same result?

Not all operators are associative:

- $(4-2)-2 \neq 4-(2-2)$

Ambiguities may have subtle implications (e.g. related to evaluation order, side effects).
Always try to make your grammar unambiguous!

Enforcing associativity

But $+$ is associative, e.g.

- As $(1 + 1) + 1 = 3$?
- As $1 + (1 + 1) = 3$?

so **who cares**, if it gives the same result?

Not all operators are associative:

- $(4 - 2) - 2 \neq 4 - (2 - 2)$

Ambiguities may have subtle implications (e.g. related to evaluation order, side effects).
Always try to make your grammar unambiguous!

Usually, subtraction, division, addition and multiplication are **left-associative** operators, while exponentiation is **right-associative**

Enforcing associativity

Modern parsers allow one to specify associativity of operators, but we can also do it directly in the grammar.

Left-associativity can be enforced by allowing **recursion on the left** only. In our example:

$$E \quad \rightarrow \quad E + E \quad | \quad N \quad | \quad (E)$$

is transformed by "expanding" the rhs of $+$:

$$E \quad \rightarrow \quad E + N \quad | \quad E + (E) \quad | \quad N \quad | \quad (E)$$

equivalently:

$$\begin{array}{lcccc} E & \rightarrow & E + E' & | & E' \\ E' & \rightarrow & N & | & (E) \end{array}$$

Enforcing associativity

General idea to enforce left-associativity of a binary operator $A \bullet A$:

1. Take the entire set of productions of A: $A \rightarrow A \bullet A \mid \gamma_1 \mid .. \mid \gamma_n$
2. Replace the production $A \rightarrow A \bullet A$ by the set of productions $A \rightarrow A \bullet \gamma_i$

Equivalently, using an auxiliary symbol:

1. Create a new non-terminal symbol A' and "move" all productions $A \rightarrow \gamma_i$ there, i.e. $A' \rightarrow \gamma_1 \mid .. \mid \gamma_n$
2. In the remaining production for A ($A \rightarrow A \bullet A$) replace the right argument by A': $A \rightarrow A \bullet A'$
3. Add the production $A \rightarrow A'$

NOTE: This idea may not always work (e.g. if $\gamma_i = A$).

Enforcing associativity

General idea to enforce left-associativity of a binary operator $A \bullet A$:

1. Take the entire set of productions of A: $A \rightarrow A \bullet A \mid \gamma_1 \mid .. \mid \gamma_n$
2. Replace the production $A \rightarrow A \bullet A$ by the set of productions $A \rightarrow A \bullet \gamma_i$

Equivalently, using an auxiliary symbol:

1. Create a new non-terminal symbol A' and "move" all productions $A \rightarrow \gamma_i$ there, i.e. $A' \rightarrow \gamma_1 \mid .. \mid \gamma_n$
2. In the remaining production for A ($A \rightarrow A \bullet A$) replace the right argument by A': $A \rightarrow A \bullet A'$
3. Add the production $A \rightarrow A'$

NOTE: This idea may not always work (e.g. if $\gamma_i = A$).

Question: How would you enforce right-associativity?

Enforcing operator precedence

Let's face now the problem of precedence in this grammar:

$$E \quad \rightarrow \quad N \quad | \quad E + E \quad | \quad E \times E \quad | \quad (E)$$

We can enforce operator priorities by stratifying the grammar into priority levels: one for each operator:

- E_0: Min priority level (sums build with $+$)
- E_1: Max priority level (products build with $\times$)

Enforcing operator precedence

Let's face now the problem of precedence in this grammar:

$$E \quad \rightarrow \quad N \quad | \quad E + E \quad | \quad E \times E \quad | \quad (E)$$

We can enforce operator priorities by stratifying the grammar into priority levels: one for each operator:

- E_0: Min priority level (sums build with $+$)
- E_1: Max priority level (products build with $\times$)

$$\begin{array}{lclclcl} E_0 & \rightarrow & E_0 + E_0 & | & E_1 & & \\ E_1 & \rightarrow & E_1 \times E_1 & | & N & | & (E_0) \end{array}$$

Note that E_1-terms cannot have an addition—unless protected under parentheses.
This prevents us from **breaking up** a multiplication with an addition.

Enforcing operator precedence

Let's face now the problem of precedence in this grammar:

$$E \quad \to \quad N \quad | \quad E + E \quad | \quad E \times E \quad | \quad (E)$$

We can enforce operator priorities by stratifying the grammar into priority levels: one for each operator:

- E_0: Min priority level (sums build with $+$)
- E_1: Max priority level (products build with $\times$)

$$\begin{array}{lcllllll} E_0 & \to & E_0 + E_0 & | & E_1 & & \\ E_1 & \to & E_1 \times E_1 & | & N & | & (E_0) \end{array}$$

Note that E_1-terms cannot have an addition—unless protected under parentheses.
This prevents us from **breaking up** a multiplication with an addition.

Example: Let us try the expression $1 + 1 \times 0$

Enforcing operator precedence

General idea of stratification to enforce precedence for operators to build terms of syntactic category A_p

1. Take the entire set of productions for A_p: $A_p \rightarrow A_p \bullet A_p \mid \gamma_1 \mid .. \mid \gamma_n$ where $\bullet$ is the binary operator that should have the lowest precedence.
2. Introduce a new symbol A_{p+1}.
3. Remove all productions $A_p \rightarrow \gamma_1 \mid .. \mid \gamma_n$
4. Add the production $A_p \rightarrow A_{p+1}$ (you can think of this as a sort of casting).
5. Add productions $A_{p+1} \rightarrow \gamma_1' \mid .. \mid \gamma_n'$ where
   - if the left-most or right-most symbols of γ_i are A_p (e.g. as happens often with binary and unary operators), then γ_i' is like γ_i where all occurrences of A_p are replaced by A_{p+1}.
   - otherwise γ_i' is just γ_i
6. Apply the same idea to A_{p+1} if it contains two or more operators that need different precedence.

Ambiguous Grammars

Now we can take the original grammar:

$$\begin{array}{lclclclcl} E & \rightarrow & N & \mid & E + E & \mid & E \times E & \mid & (E) \\ N & \rightarrow & 0 & \mid & 1 & \mid & N0 & \mid & N1 \end{array}$$

And apply the transformations we have seen:

1. Enforce operator priorities by stratification.
2. Enforce associativity by allowing recursion on one side only.

To obtain:

$$\begin{array}{lclclclcl} E_0 & \rightarrow & E_0 + E_1 & \mid & E_1 & & & & \\ E_1 & \rightarrow & E_1 \times E_1' & \mid & E_1' & & & & \\ E_1' & \rightarrow & N & \mid & (E_0) & & & & \\ N & \rightarrow & 0 & \mid & 1 & \mid & N0 & \mid & N1 \end{array}$$

Exercise 2.3: ambiguous boolean expressions?

Consider the following simple grammar for boolean expressions:

$$B \quad \rightarrow \quad \texttt{true} \mid B \ \texttt{or} \ B \mid \texttt{not} \ B \mid (B)$$

Exercise: Show that it is ambiguous by providing examples and design an equivalent grammar that solves the ambiguities of the examples.

Left-recursion and left-associativity

Restricting to left-recursion is a way to enforce left-associativity:

$$E \to E + N \mid N$$

Left-recursion and left-associativity

Restricting to left-recursion is a way to enforce left-associativity:

$$E \rightarrow E + N \mid N$$

Unfortunately some parsing algorithms struggle with it:

```
parse s as E = {
  lhs = parse s as E;
  read '+';
  rhs = parse s as N;
}
```

Left-recursion and left-associativity

Restricting to left-recursion is a way to enforce left-associativity:

$$E \to E + N \mid N$$

Unfortunately some parsing algorithms struggle with it:

```
parse s as E = {
  lhs = parse s as E;
  read '+';
  rhs = parse s as N;
}
```

That leads immediately to infinite recursion!

Left-recursion and left-associativity

The problem is that our grammar is **left recursive**!

Left-recursion and left-associativity

The problem is that our grammar is **left recursive**!
We could make it **right recursive** instead:

$$E \quad \rightarrow \quad N + E \mid N$$

Left-recursion and left-associativity

The problem is that our grammar is **left recursive**!
We could make it **right recursive** instead:

$$E \quad \rightarrow \quad N + E \mid N$$

...but that makes our operators right-associative!

Left-recursion and left-associativity

The problem is that our grammar is **left recursive**!
We could make it **right recursive** instead:

$$E \quad \rightarrow \quad N + E \mid N$$

...but that makes our operators right-associative!

Left-recursion and left-associativity

Solution: notice that an expression is really just a term followed by a **list** of zero or more additions.

Left-recursion and left-associativity

Solution: notice that an expression is really just a term followed by a **list** of zero or more additions.

$$E \quad \rightarrow \quad N\ (+\ N)^*$$

Left-recursion and left-associativity

Solution: notice that an expression is really just a term followed by a **list** of zero or more additions.

$$E \quad \rightarrow \quad N\ (+\ N)^*$$

where we use actually a regular expression in a grammar rule for abbreviation. Without this abbreviation:

$$\begin{array}{lcl} E & \rightarrow & N\ E' \\ E' & \rightarrow & +\ N\ E' \mid \epsilon \end{array}$$

Left-recursion and left-associativity

Solution: notice that an expression is really just a term followed by a **list** of zero or more additions.

$$E \quad \rightarrow \quad N\ (+\ N)^*$$

where we use actually a regular expression in a grammar rule for abbreviation. Without this abbreviation:

$$\begin{array}{lcl} E & \rightarrow & N\ E' \\ E' & \rightarrow & +\ N\ E' \mid \epsilon \end{array}$$

In this form we can enforce left-associativity without left-recursion.

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$
\begin{aligned}
L &= L_1 \cup L_2 \\
L_1 &= \{\, a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \,\} \\
L_2 &= \{\, a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \,\}
\end{aligned}
$$

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$
\begin{aligned}
L &= L_1 \cup L_2 \\
L_1 &= \{\, a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \,\} \\
L_2 &= \{\, a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \,\}
\end{aligned}
$$

Words in L must have

- the same number of a's and b's and the same number of c's and d's,
  or
- the same number of a's and d's and the same number of b's and c's.

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$\begin{aligned} L &= L_1 \cup L_2 \\ L_1 &= \{\, a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \,\} \\ L_2 &= \{\, a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \,\} \end{aligned}$$

Intuition:

- We can write unambiguous grammars for L_1 and L_2.

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$
\begin{aligned}
L &= L_1 \cup L_2 \\
L_1 &= \{\, a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \,\} \\
L_2 &= \{\, a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \,\}
\end{aligned}
$$

Intuition:

- We can write unambiguous grammars for L_1 and L_2.
- The union is ambiguous however: all words $a^n b^n c^n d^n$ can be derived both from L_1 and L_2.

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$
\begin{aligned}
L &= L_1 \cup L_2 \\
L_1 &= \{\, a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \,\} \\
L_2 &= \{\, a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \,\}
\end{aligned}
$$

Intuition:

- We can write unambiguous grammars for L_1 and L_2.
- The union is ambiguous however: all words $a^n b^n c^n d^n$ can be derived both from L_1 and L_2.
- There is no way to resolve this with a different construction. (Hard to prove)

Ambiguous Grammars

Can we always resolve the ambiguity in a grammar?

We cannot! Some languages are **inherently ambiguous**.

Consider the following language:

$$\begin{aligned} L &= L_1 \cup L_2 \\ L_1 &= \{ a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \} \\ L_2 &= \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \} \end{aligned}$$

Intuition:

- We can write unambiguous grammars for L_1 and L_2.
- The union is ambiguous however: all words $a^n b^n c^n d^n$ can be derived both from L_1 and L_2.
- There is no way to resolve this with a different construction. (Hard to prove)

No grammar for L is unambiguous!

Exercise Session

- Finish the exercises 2.1, 2.2 and 2.3;
- Solve the exercises of the following slides;
- If not yet done, read the description of the mandatory assignment.

Exercise 2.4: more HTML

Consider the grammar for HTML in the figure below

1. *Char* → $a \mid A \mid \cdots$
2. *Text* → $\epsilon \mid$ *Char Text*
3. *Doc* → $\epsilon \mid$ *Element Doc*
4. *Element* → *Text* |
<EM> *Doc* </EM> |
<P> *Doc* |
<OL> *List* </OL> | ⋯
5. *ListItem* → <LI> *Doc*
6. *List* → $\epsilon \mid$ *ListItem List*

Figure 5.13: Part of an HTML grammar

Exercise 2.4: more HTML

Solve the following exercise from the book.

Exercise 5.3.4: Add the following forms to the HTML grammar of Fig. 5.13:

* a) A list item must be ended by a closing tag `</LI>`.

b) An element can be an unordered list, as well as an ordered list. Unordered lists are surrounded by the tag `<UL>` and its closing `</UL>`.

! c) An element can be a table. Tables are surrounded by `<TABLE>` and its closer `</TABLE>`. Inside these tags are one or more rows, each of which is surrounded by `<TR>` and `</TR>`. The first row is the header, with one or more fields, each introduced by the `<TH>` tag (we'll assume these are not closed, although they should be). Subsequent rows have their fields introduced by the `<TD>` tag.

Exercise 2.5: from DTD to CFG

Solve the following exercise from the book.

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT PROF (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)> ]>
```

Figure 5.16: A DTD for courses

Exercise 5.3.5: Convert the DTD of Fig. 5.16 to a context-free grammar.

You don't need to specify `#PCDATA`.

Exercise 2.6: a simple ambiguous grammar

Solve the following exercises from the book:

* **Exercise 5.4.1:** Consider the grammar

$$S \to aS \mid aSbS \mid \epsilon$$

This grammar is ambiguous. Show in particular that the string aab has two:

a) Parse trees.

b) Leftmost derivations.

c) Rightmost derivations.

d) Is there a construct in standard programming languages that could have a similar ambiguity?

*! **Exercise 5.4.3:** Find an unambiguous grammar for the language of Exercise 5.4.1.

Exercise 2.7: Polish expressions

Exercise 5.4.7: The following grammar generates *prefix* expressions with operands x and y and binary operators $+$, $-$, and $*$:

$$E \to +EE \mid *EE \mid -EE \mid x \mid y$$

a) Find leftmost and rightmost derivations, and a derivation tree for the string +*-xyxy.

b) Give a parse tree for the expression in (a).

c) Is this grammar ambiguous? If yes, provide an example. Otherwise, provide an informal argument.