Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2023 - 2024

David van Balen

Department of Information and Computing Sciences
Utrecht University

2023-11-13

1. Introduction

Course Content Overview

Languages . . .

A **language** is a set of “correct” sentences.

- But what does that mean?

Languages . . .

A **language** is a set of "correct" sentences.

- But what does that mean?
- What is the difference between natural and formal languages?

Languages . . .

A **language** is a set of “correct” sentences.

- But what does that mean?
- What is the difference between natural and formal languages?
- Are all languages equally difficult or complicated?

Languages . . .

A **language** is a set of “correct” sentences.

- But what does that mean?
- What is the difference between natural and formal languages?
- Are all languages equally difficult or complicated?
- How can one decide whether a sentence is correct?

Languages . . .

A **language** is a set of "correct" sentences.

- But what does that mean?
- What is the difference between natural and formal languages?
- Are all languages equally difficult or complicated?
- How can one decide whether a sentence is correct?
- How can one represent a correct sentence?

. . . and compilers

A **compiler** translates one language into another (possibly the same). How?

- get hold of the structure of the input program

. . . and compilers

A **compiler** translates one language into another (possibly the same). How?

- get hold of the structure of the input program
- attach semantics to a sequence of symbols

. . . and compilers

A **compiler** translates one language into another (possibly the same). How?

- get hold of the structure of the input program
- attach semantics to a sequence of symbols
- check whether a program makes sense

. . . and compilers

A **compiler** translates one language into another (possibly the same). How?

- get hold of the structure of the input program
- attach semantics to a sequence of symbols
- check whether a program makes sense
- optimize

. . . and compilers

A **compiler** translates one language into another (possibly the same). How?

- get hold of the structure of the input program
- attach semantics to a sequence of symbols
- check whether a program makes sense
- optimize
- generate good machine code

Languages, grammars, and meaning

Computer science studies information processing.

- We describe and transfer **information** by means of **language**

Languages, grammars, and meaning

Computer science studies information processing.

- We describe and transfer **information** by means of **language**
- Information is obtained by assigning **meaning** to **sentences**

Languages, grammars, and meaning

Computer science studies information processing.

- We describe and transfer **information** by means of **language**
- Information is obtained by assigning **meaning** to **sentences**
- The **meaning** of a sentence is inferred from its **structure**

Languages, grammars, and meaning

Computer science studies information processing.

- We describe and transfer **information** by means of **language**
- Information is obtained by assigning **meaning** to **sentences**
- The **meaning** of a sentence is inferred from its **structure**
- The **structure** of a sentence is described by means of a **grammar**

Languages, grammars, and meaning

Computer science studies information processing.

- We describe and transfer **information** by means of **language**
- Information is obtained by assigning **meaning** to **sentences**
- The **meaning** of a sentence is inferred from its **structure**
- The **structure** of a sentence is described by means of a **grammar**
- Getting this wrong is a common source of security bugs!

In this course

- Classes ("difficulty levels") of languages

In this course

- Classes ("difficulty levels") of languages
  - context-free languages

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification

In this course

- Classes (“difficulty levels”) of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers
- Parsing context-free and regular languages, using

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers
- Parsing context-free and regular languages, using
  - parser combinators

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers
- Parsing context-free and regular languages, using
  - parser combinators
  - parser generators

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers
- Parsing context-free and regular languages, using
  - parser combinators
  - parser generators
  - finite state automata

In this course

- Classes ("difficulty levels") of languages
  - context-free languages
  - regular languages
- Describing languages formally, using
  - grammars
  - finite state automata
- Grammar transformations
  - for simplification
  - for obtaining more efficient parsers
- Parsing context-free and regular languages, using
  - parser combinators
  - parser generators
  - finite state automata
- How to go from syntax to semantics

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;
- To **parse**, i.e., to recognise (build) such structures in (from) a sequence of symbols;

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;
- To **parse**, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- To **analyse** grammars to see whether or not specific properties hold;

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;
- To **parse**, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- To **analyse** grammars to see whether or not specific properties hold;
- To **compose** components such as parsers, analysers, and code generators;

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;
- To **parse**, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- To **analyse** grammars to see whether or not specific properties hold;
- To **compose** components such as parsers, analysers, and code generators;
- To **apply these techniques** in the construction of all kinds of programs;

Learning goals

- To **describe** structures (i.e., "formulas") using **grammars**;
- To **parse**, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- To **analyse** grammars to see whether or not specific properties hold;
- To **compose** components such as parsers, analysers, and code generators;
- To **apply these techniques** in the construction of all kinds of programs;
- To **explain** and **prove** why certain problems can or cannot be described by means of formalisms such as context-free grammars or finite-state automata.

1.2 Course Organization

Course website

ics.uu.nl/docs/vakken/b3tc

Assignments

Three practicals:

- P0: refresh your FP, doesn't count for the final grade
- P1–P3: theoretical and practical aspects
- Work in groups of two, self organize

Exams

Two exams: T1, T2

- Contents for each is specified in the schedule
- You cannot use lecture notes or other material for the exams.

Resit (aanvullende toets) exam: T3

- You will receive an e-mail that tells you if you qualify for resit/relab, and telling you what you should in fact do. See the Osiris website for the rules.

Haskell

We use Haskell because many concepts from formal language theory have a direct correspondence in Haskell.

Haskell

We use Haskell because many concepts from formal language theory have a direct correspondence in Haskell.

Formal languages	Haskell
alphabet	datatype
sequence	list type
sentence/word	a concrete list
abstract syntax	datatype
grammar	parser
grammar transformation	parser transformation
parse tree	value of abstract syntax type
semantics	fold function, algebra

1.3 Haskell Refresh

Functions

Pattern matching

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Functions

Pattern matching and recursion.

$$\begin{aligned} &\mathsf{length} :: [\mathsf{a}] \to \mathsf{Int} \\ &\mathsf{length}\ [\,] \qquad\quad = 0 \\ &\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = 1 + \boxed{\mathsf{length\ xs}} \end{aligned}$$

Functions

Pattern matching and recursion.

$$\begin{aligned}
&\mathsf{length} :: \boxed{[\mathsf{a}] \to \mathsf{Int}} \\
&\mathsf{length}\ [\,] \qquad\quad = 0 \\
&\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = 1 + \mathsf{length}\ \mathsf{xs}
\end{aligned}$$

Type signatures, but type inference.

Functions

Pattern matching and recursion.

$$\begin{aligned} &\mathsf{length} :: [\mathsf{a}] \to \mathsf{Int} \\ &\mathsf{length}\ []\ \ \ \ \ \ \ \ \ \ = 0 \\ &\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = 1 + \mathsf{length}\ \mathsf{xs} \end{aligned}$$

Type signatures, but type inference.

Polymorphism – length works for any list.

Currying

Functions with multiple arguments are written as "functions to functions to functions ...":

$$(\mathbin{+\!\!\!+}) :: [a] \to [a] \to [a]$$
$$[] \qquad \mathbin{+\!\!\!+} ys = ys$$
$$(x : xs) \mathbin{+\!\!\!+} ys = x : (xs \mathbin{+\!\!\!+} ys)$$

Again, $(\mathbin{+\!\!\!+})$ is polymorphic. We need not know the type of list elements, but both argument lists must have the same type of elements!

Higher-order functions – map

Applying a function to every element of a list:

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Higher-order functions – map

Applying a function to every element of a list:

map :: (a → b) → [a] → [b]

Example:

map $(+1)$ $[1, 2, 3, 4, 5]$
$= \quad [2, 3, 4, 5, 6]$

Higher-order functions – filter

Filtering a list according to a predicate:

$$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

Higher-order functions – filter

Filtering a list according to a predicate:

$$\text{filter} :: (\text{a} \rightarrow \text{Bool}) \rightarrow [\text{a}] \rightarrow [\text{a}]$$

Example:

$$\begin{aligned} &\text{filter even } [1, 2, 3, 4, 5] \\ &\quad = [\;\; 2, \;\; 4 \;\;] \end{aligned}$$

Higher-order functions – foldr

Traversing a list according to its structure:

$$\text{foldr} :: (a \to b \to b) \to b \to [a] \to b$$

Higher-order functions – foldr

Traversing a list according to its structure:

$$\mathsf{foldr} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b}$$

Example:

$$\begin{aligned}
&\mathsf{foldr}\ (+)\ 0\ [1, \quad 2, \quad 3, \quad 4, \quad 5\ ] \\
=\ &\mathsf{foldr}\ (+)\ 0\ (1\ :\ 2\ :\ 3\ :\ 4\ :\ 5\ :\ [\,]) \\
=\ &\mathsf{foldr}\ (+)\ 0\ (1\ :\ (2\ :\ (3\ :\ (4\ :\ (5\ :\ [\,]))))) \\
=\ &\qquad 1 + (2 + (3 + (4 + (5 + 0)))) \\
=\ &\qquad 15
\end{aligned}$$

Datatypes

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Datatypes

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Datatypes can have **parameters**.

Datatypes

$$\textbf{data}\ \mathsf{Tree\ a} = \mathsf{Leaf\ a} \mid \mathsf{Node\ (Tree\ a)\ (Tree\ a)}$$

Datatypes can have **parameters**.

Multiple **constructors**:

$$\begin{aligned}\mathsf{Leaf} &:: \mathsf{a} \to \mathsf{Tree\ a}\\ \mathsf{Node} &:: \mathsf{Tree\ a} \to \mathsf{Tree\ a} \to \mathsf{Tree\ a}\end{aligned}$$

Constructors describe the shape of values of the datatype. They can be used in patterns.

Functions on trees

$$
\begin{aligned}
&\mathsf{size} :: \mathsf{Tree\ a} \rightarrow \mathsf{Int}\\
&\mathsf{size}\ (\mathsf{Leaf\ x}) \quad = 1\\
&\mathsf{size}\ (\mathsf{Node\ l\ r}) = \mathsf{size\ l} + \mathsf{size\ r}
\end{aligned}
$$

Functions on trees

```
size :: Tree a → Int
size (Leaf x)   = 1
size (Node l r) = size l + size r
```

Exercise: A function that reverses (mirrors) a tree.

Functions on trees

$$\text{size} :: \text{Tree a} \rightarrow \text{Int}$$
$$\text{size (Leaf x)} \quad = 1$$
$$\text{size (Node l r)} = \text{size l} + \text{size r}$$

Exercise: A function that reverses (mirrors) a tree.

$$\text{reverse} :: \text{Tree a} \rightarrow \text{Tree a}$$
$$\text{reverse (Leaf x)} \quad = \text{Leaf x}$$
$$\text{reverse (Node l r)} = \text{Node (reverse r) (reverse l)}$$

1.4 (Formal) Languages

What is a language?

What is a language?

A language is a set of sentences (or words).

What is a language?

A language is a set of sentences (or words).

Which sentences belong to a language, and why?

- In natural languages, this is often informally defined and subject to discussion.

What is a language?

A language is a set of sentences (or words).

Which sentences belong to a language, and why?

- In natural languages, this is often informally defined and subject to discussion.
- For a formal language, we want a precise definition.

Sets

A **set** is a collection of elements.

- No duplicates
- No order
- The empty set: $\emptyset$
- A nonempty set: $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

Sets

A **set** is a collection of elements.

- No duplicates
- No order
- The empty set: $\emptyset$
- A nonempty set: $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

- Union
- Intersection

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\{\mathtt{0}, \mathtt{1}\}$

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\{\mathtt{0}, \mathtt{1}\}$
- The set of all Latin letters

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{a, b, c\}$
- $\{0, 1\}$
- The set of all Latin letters
- The set of ASCII characters

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\{\mathtt{0}, \mathtt{1}\}$
- The set of all Latin letters
- The set of ASCII characters
- The set of Unicode code points

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\{\mathtt{0}, \mathtt{1}\}$
- The set of all Latin letters
- The set of ASCII characters
- The set of Unicode code points
- $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- { `a`, `b`, `c` }
- { `0`, `1` }
- The set of all Latin letters
- The set of ASCII characters
- The set of Unicode code points
- { `A`, `C`, `G`, `T` }
- { `if`, `then`, `else`, `do` }

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\{\mathtt{0}, \mathtt{1}\}$
- The set of all Latin letters
- The set of ASCII characters
- The set of Unicode code points
- $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
- $\{\mathtt{if}, \mathtt{then}, \mathtt{else}, \mathtt{do}\}$
- $\{\mathtt{+}, \mathtt{-}\}$

Alphabet

An **alphabet** is a (finite) set of symbols that can be used to form sentences.

- $\{\texttt{a}, \texttt{b}, \texttt{c}\}$
- $\{\texttt{0}, \texttt{1}\}$
- The set of all Latin letters
- The set of ASCII characters
- The set of Unicode code points
- $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$
- $\{\texttt{if}, \texttt{then}, \texttt{else}, \texttt{do}\}$
- $\{\texttt{+}, \texttt{-}\}$
- 

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Examples of sequences over A:

- abc

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{a, b, c\}$. Examples of sequences over A:

- abc
- a

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{a, b, c\}$. Examples of sequences over A:

- abc
- a
- acccabcabcabbaca

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Examples of sequences over A:

- abc
- a
- acccabcabcabbaca
- bbbbbbbb

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{a, b, c\}$. Examples of sequences over A:

- abc
- a
- acccabcabcabbaca
- bbbbbbbb
- ε

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Examples of sequences over A:

- `abc`
- `a`
- `acccabcabcabbaca`
- `bbbbbbbb`
- ε

Sequence

Given a set, we can consider (finite) sequences of elements of that set.

Let $\mathsf{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Examples of sequences over A:

- `abc`
- `a`
- `acccabcabcabbaca`
- `bbbbbbbb`
- ε

The empty sequence is difficult to visualize. Therefore, we usually write ε as a placeholder to denote the empty sequence.

Sequences, inductively

Given an arbitrary sequence over elements of a set A, we can make one of the two following observations:

- it is the empty sequence ε,

Sequences, inductively

Given an arbitrary sequence over elements of a set A, we can make one of the two following observations:

- it is the empty sequence ε,
- the sequence has a first element $a \in A$, and if we split off that element, the tail is still a (possibly empty) sequence z.

Sequences, inductively

Given an arbitrary sequence over elements of a set A, we can make one of the two following observations:

- it is the empty sequence ε,
- the sequence has a first element $a \in A$, and if we split off that element, the tail is still a (possibly empty) sequence z.

Sequences, inductively

Given an arbitrary sequence over elements of a set A, we can make one of the two following observations:

- it is the empty sequence ε,
- the sequence has a first element $a \in A$, and if we split off that element, the tail is still a (possibly empty) sequence z.

We can use this observation to define sequences.

Sequences, inductively

Given a set A. The set of **sequences over** A, written A^*, is defined as follows:

- the empty sequence ε is in A^*,

Sequences, inductively

Given a set A. The set of **sequences over** A, written A^*, is defined as follows:

- the empty sequence ε is in A^*,
- if $a \in A$ and $z \in A^*$, then az is in A^*.

Sequences, inductively

Given a set A. The set of **sequences over A**, written A^*, is defined as follows:

- the empty sequence ε is in A^*,
- if $a \in A$ and $z \in A^*$, then az is in A^*.

Sequences, inductively

Given a set A. The set of **sequences over** A, written A^*, is defined as follows:

- the empty sequence ε is in A^*,
- if $a \in A$ and $z \in A^*$, then az is in A^*.

In such an inductive definition, it is implicitly understood that

- nothing else is in A^*,

Sequences, inductively

Given a set A. The set of **sequences over** A, written A^*, is defined as follows:

- the empty sequence ε is in A^*,
- if $a \in A$ and $z \in A^*$, then az is in A^*.

In such an inductive definition, it is implicitly understood that

- nothing else is in A^*,
- we can only apply the construction steps a finite number of times, i.e., only finite sequences are in A^*.

Remarks about sequences

- How many elements does $\emptyset$ contain?

Remarks about sequences

- How many elements does $\emptyset$ contain?
- How many elements does $\emptyset^*$ contain?

Remarks about sequences

- How many elements does $\emptyset$ contain?
- How many elements does $\emptyset^*$ contain?
- How many elements does $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ contain?

Remarks about sequences

- How many elements does $\emptyset$ contain?
- How many elements does $\emptyset^*$ contain?
- How many elements does $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ contain?
- How many elements does $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ contain?

Language

Given an alphabet A, a **language** is a subset of A^*.

Language

Given an alphabet A, a **language** is a subset of A^*.

Note that we consider any set X to be a subset of itself: $X \subseteq X$.

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Language

Given an alphabet A, a **language** is a subset of A^*.

Note that we consider any set X to be a subset of itself: $X \subseteq X$.

So A^* is a valid language with alphabet A.

How to define a language?

So a language is just the set of correct sentences.

How to define a language?

So a language is just the set of correct sentences.

But how do we define such a set?

- By enumerating all elements?
- By using a predicate?
- By giving an inductive definition?
- ...

How to define a language?

So a language is just the set of correct sentences.

But how do we define such a set?

- By enumerating all elements?
- By using a predicate?
- By giving an inductive definition?
- ...

All these are possible, and more.

Example

Let the set of digits $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be our alphabet.

Example

Let the set of digits $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be our alphabet.

This is a language:

> $L = \{2, 3, 5, 7, 11, 13, 17, 19\}$

How can we describe this language?

Example

Let the set of digits $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be our alphabet.

This is a language:

$L = \{2, 3, 5, 7, 11, 13, 17, 19\}$

How can we describe this language?

- The language L is the language over D of all prime numbers less than 20.

Languages by enumeration

- Enumerating all elements of a language is impossible if the language is infinite.

Languages by enumeration

- Enumerating all elements of a language is impossible if the language is infinite.
- Most interesting languages are infinite:
  - C#
  - Haskell
  - ...

Languages by enumeration

- Enumerating all elements of a language is impossible if the language is infinite.
- Most interesting languages are infinite:
  - C#
  - Haskell
  - ...
- Defining a language using a predicate seems better.

Defining by predicate example

Let $A = \{ \mathtt{a}, \mathtt{b}, \mathtt{c} \}$ be our alphabet.

Defining by predicate example

Let $A = \{a, b, c\}$ be our alphabet.

Then

> $PAL = \{s \in A^* \mid s = s^R\}$

is the language of **palindromes** over A.

Example – contd.

Palindromes can also be defined inductively:

- ε is in PAL,
- a, b, c are in PAL,
- if P is in PAL, then aPa, bPb and cPc are also in PAL.

By predicate vs. by induction

Which definition is better?

> $\mathsf{PAL} = \{\mathsf{s} \in \mathsf{A}^* \mid \mathsf{s} = \mathsf{s}^R\}$

or

The set PAL of palindromes over A is defined as follows:

- ε is in PAL,
- a, b, c are in PAL,
- if P is in PAL, then aPa, bPb and cPc are also in PAL.

By predicate vs. by induction

Definition by predicate is (in this case) shorter.

By predicate vs. by induction

Definition by predicate is (in this case) shorter.

How can we check whether a given sequence is in PAL?

By predicate vs. by induction

Definition by predicate is (in this case) shorter.

How can we check whether a given sequence is in PAL?

How can we generate all the words in PAL?

By predicate vs. by induction

Definition by predicate is (in this case) shorter.

How can we check whether a given sequence is in PAL?

How can we generate all the words in PAL?

An inductive definition gives us more structure, and makes it easier to explain **why** a sentence is in the language.

Summary

Alphabet A finite set of symbols.

This werkcollege: Haskell setup and P0.

Summary

Alphabet A finite set of symbols.

Language A set of words/sentences, i.e., sequences of symbols from the alphabet.

This werkcollege: Haskell setup and P0.

Summary

Alphabet A finite set of symbols.

Language A set of words/sentences, i.e., sequences of symbols from the alphabet.

Grammar Next lecture: A way to define a language inductively by means of rewrite rules.

This werkcollege: Haskell setup and P0.