Universiteit Utrecht

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

Talen en Compilers

2010/2011, periode 2

Johan Jeuring

Department of Information and Computing Sciences
Utrecht University

November 15, 2010

1. Introduction

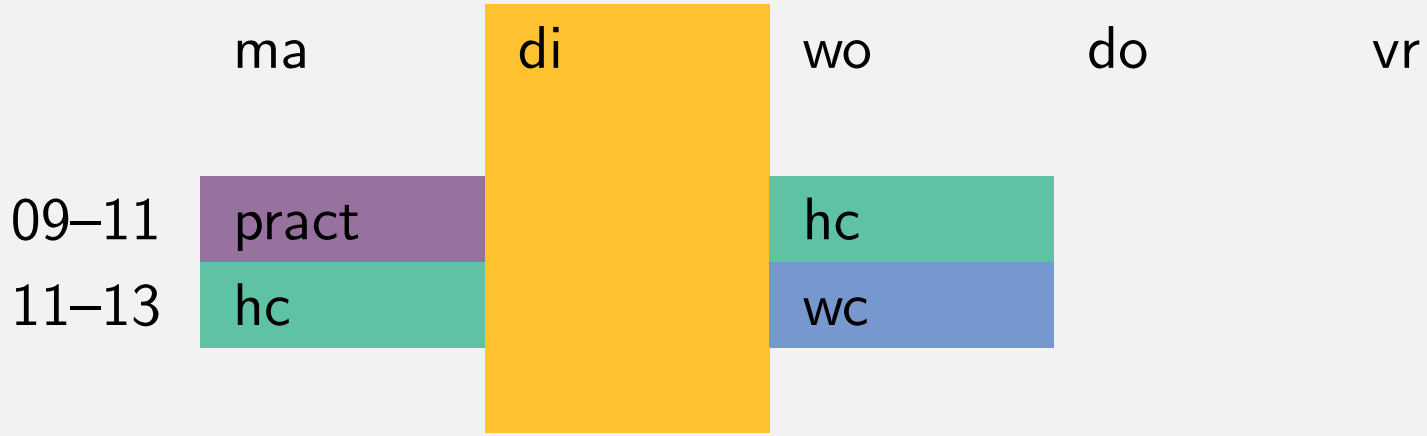
This lecture

Introduction

- Organization
- Course overview
- Haskell recap
- (Formal) Languages

1.1 Organization

Weekly schedule



lecture

exercise session: **no computers**

labs: use computers

dinsdag: deadline for lab submissions

People

Teacher:

- Johan Jeuring (`johanj@cs.uu.nl`, BBL571)

Assistants:

- Bram Schuur
- Jeroen Goudsmit

Exercise classes + labs

4 labs (P0, P1, P2, P3)

- P0: introduce and refresh FP, doesn't count for the final grade
- P1-P3: theoretical and practical aspects
- teams of two, but:
  - every student should be able to explain **all** his (her) solutions

Self- and peer-assessment

We will use **self- and peer-assessment** for the labs. Why?

- Self- and peer-assessment give you first-hand, active involvement with the criteria for good learning of the subject
- You learn how to select good evidence
- Judging whether a performance or product meets given criteria is vital for effective professional action

Organization of self- and peer-assessment

- Every group hands in a solution via submit
- Every student receives a worked-out solution from an assistant
- Every group assesses its own work (via submit)
- Every student assesses the work of a fellow group (via submit)
- The assistants will randomly assess some submissions
- If the grades are not more than 1 apart, we will take the higher grade
- Otherwise the assistant will determine the grade

Exams

Two exams (T1, T2).

- T1: monday, 13 december 2010, 10:30 – 13
- T2: monday, 31 january 2011, 8:30 – 11:30
- You cannot use lecture notes or other material
- The second exam is about the complete material

Grading

$$\mathsf{T} = 0.5 * \mathsf{T1} + 0.5 * \mathsf{T2}$$
$$\mathsf{P} = 0.3 * \mathsf{P1} + 0.3 * \mathsf{P2} + 0.4 * \mathsf{P3}$$
$$\mathsf{finalgrade} = \mathsf{round}\ (\textbf{if}\ \mathsf{P} \geqslant 5 \wedge \mathsf{T} \geqslant 5\ \textbf{then}\ (\mathsf{T} + \mathsf{P})\ /\ 2\ \textbf{else}\ \ \mathsf{min}\ \mathsf{P}\ \mathsf{T})$$

Rounding to halves above 6, and to whole numbers below 6, so 5.5 becomes 6 and 5.4 becomes 5.

If one of the five grades is missing, no grade will be given, and the course is 'incomplete'.

Herkansing/resit

Eventueel kan er een een aanvullende toets worden gedaan (in de herkansingsweek halverwege de periode volgend op het vak). Je kunt hierbij T1 of T2 herkansen, maar niet allebei (om een onvoldoende of een ontbrekend cijfer te vervangen). De aanvullende toets gaat over de gehele stof. Als aanvullende toets voor het practicum kan er een extra practicumopdracht worden gemaakt, die je kunt inzetten voor een van P1, P2 en P3 (om een onvoldoende of een ontbrekend cijfer te vervangen).

Wiki and IRC

`http://www.cs.uu.nl/wiki/TC/`

There is an irc channel for the course:

- `#tc2010` on `irc.freenode.net`
- I will be there often, but not always...

Lecture notes

Languages and Compilers (version 2010)

- Online via Wiki.
- Available for sale at OSZ from next week on (I will announce availability)
- There are small changes compared with last year.
- Work in progress – feedback welcome!

Software

We use:

Software

We use: **Haskell**

Software

We use: **Haskell**

What you need:

- GHC (Glasgow Haskell Compiler), version 6.8.3 or later
- Alex (lexer generator)
- Happy (parser generator)
- uu-tc (course-specific library)

Software

We use: **Haskell**

What you need:

- GHC (Glasgow Haskell Compiler), version 6.8.3 or later
- Alex (lexer generator)
- Happy (parser generator)
- uu-tc (course-specific library)

Haskell Platform (contains GHC, Alex, Happy, and more)

`http://hackage.haskell.org/platform`

1.2 Course overview

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

Many of these topics will be handled in much more detail in the master course **Compiler Construction** (part of the Computing Science master program).

Languages, grammars, meaning, and life

Computer science studies information processing.

Languages, grammars, meaning, and life

Computer science studies information processing.

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

Languages, grammars, meaning, and life

Computer science studies information processing.

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

Languages, grammars, meaning, and life

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, meaning, and life

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, meaning, and life

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**
- Grammars and languages are essential for many human activities: verbal and written communication, musical scores, dna, but also programming

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**;
- 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 understand the concept of **compositionality**;
- to **apply these techniques** in the construction of all kinds of programs;
- to familiarise oneself with the concept of **computability**.

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 recap

Functions

Pattern matching

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

Functions

Pattern matching and recursion.

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

Functions

Pattern matching and recursion.

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

Type signatures, but type inference.

Functions

Pattern matching and recursion.

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

Type signatures, but type inference.

Polymorphism – length works for any list.

Currying

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

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[] \quad\quad +\!\!+ ys = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

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

Higher-order functions – map

Applying a function to every element of a list:

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

Example:

$$\begin{aligned} \mathsf{map}\ (+1)\ [1,2,3,4,5] \\ = \quad [2,3,4,5,6] \end{aligned}$$

Higher-order functions – filter

Filtering a list according to a predicate:

```
filter :: (a → Bool) → [a] → [a]
```

Example:

```
filter even [1, 2, 3, 4, 5]
        =   [  2,   4  ]
```

Higher-order functions – foldr

Traversing a list according to its structure:

```
foldr :: (a → b → b) → b → [a] → b
```

Example:

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

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**.

Multiple **constructors**:

```
Leaf :: a → Tree a
Node :: Tree a → Tree a → Tree a
```

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

Functions on trees

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

reverse :: Tree a → Tree a
reverse (Leaf x)   = Leaf x
reverse (Node l r) = Node (reverse r) (reverse l)
```

1.4 (Formal) Languages

What is a language?

What is a language?

A language consists of sentences (or words).

What is a language?

A language consists 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.

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 characters
> $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
> $\{\mathtt{if}, \mathtt{then}, \mathtt{else}, \mathtt{do}\}$
> $\{\mathtt{+}, \mathtt{-}\}$
> $\{$  $\}$

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

> abc
>
> a
> acccabcabcabbaca
> bbbbbbbbb

The empty sequence is difficult to visualize.

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
> ε
> a
> acccabcabcabbaca
> bbbbbbbbb

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,
- 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^*,
- 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^*,
- 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?

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: $X \subseteq X$.

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.

This is a language:

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

Find other ways to 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:

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

Find other ways to describe this language!

The language L is the language over D of all prime numbers smaller than twenty.

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:
  - Java
  - Haskell
  - ...

Languages by enumeration

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

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.

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, makes it easier to explain **why** a sentence is in the language.

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.