Introduction

1. Introduction
   - Administrivia
   - Course contents

2. Regular Expressions
   - Definitions
   - Examples
   - Scanner generators

January 19th
This is an **elective course** for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- The tools you use to build pieces of software!
- You need to use them to implement solutions to some problems.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.
- **The solution** to some problems
  - you need to know how to implement them.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are
- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.
- **The solution** to some problems
  - you need to know how to implement them.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.

- **The solution** to some problems
  - you need to know how to implement them.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.

- **The solution** to some problems
  - you need to know how to implement them.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.
- **The solution** to some problems
  - you need to know how to implement them.
This is an elective course for the master programmes *Embedded Intelligent Systems* and *Computer Systems Engineering*.

Computer languages are

- **The tools** you use to build pieces of software!
  - you need to understand them properly,
  - you need to learn new ones.
- **The solution** to some problems
  - you need to know how to implement them.
Resources

Teachers

- Jerker Bengtsson
  www2.hh.se/staff/jebe
- Verónica Gaspes
  www2.hh.se/staff/vero

On Line

- Web page
  www2.hh.se/staff/jebe/languages.
- Lecture notes, notice board, project instructions.
- Manuals for the tools we use.

A good book, organized around building a compiler.

A lot of material, helps even if you want to follow an advanced course. We look at the first part, upto chapter 12.
Resources

Teachers

- Jerker Bengtsson
  www2.hh.se/staff/jebe
- Verónica Gaspes
  www2.hh.se/staff/vero

On Line

- Web page
  www2.hh.se/staff/jebe/languages.
- Lecture notes, notice board, project instructions.
- Manuals for the tools we use.

A good book, organized around building a compiler.

A lot of material, helps even if you want to follow an advanced course. We look at the first part, upto chapter 12.
Resources

Teachers

- Jerker Bengtsson
  www2.hh.se/staff/jebe
- Verónica Gaspes
  www2.hh.se/staff/vero

On Line

- Web page
  www2.hh.se/staff/jebe/languages.
- Lecture notes, notice board, project instructions.
- Manuals for the tools we use.

A good book, organized around building a compiler.

A lot of material, helps even if you want to follow an advanced course. We look at the first part, upto chapter 12.
Resources

Teachers

- Jerker Bengtsson
  www2.hh.se/staff/jebe
- Verónica Gaspes
  www2.hh.se/staff/vero

On Line

- Web page
  www2.hh.se/staff/jebe/languages.
- Lecture notes, notice board, project instructions.
- Manuals for the tools we use.

A good book, organized around building a compiler.

A lot of material, helps even if you want to follow an advanced course. We look at the first part, upto chapter 12.
Resources

Teachers

- Jerker Bengtsson
  www2.hh.se/staff/jebe
- Verónica Gaspes
  www2.hh.se/staff/vero

On Line

- Web page
  www2.hh.se/staff/jebe/languages.
- Lecture notes, notice board, project instructions.
- Manuals for the tools we use.

A good book, organized around building a compiler.

A lot of material, helps even if you want to follow an advanced course. We look at the first part, upto chapter 12.
A programming project where you implement a compiler for a subset of the Java programming language.

Small computer based exercises about formal languages.

The instructions will be on the web, including deadlines.

The project is evaluated during the examination week. If you do not pass, it can be evaluated during the following examination week.
A programming project where you implement a compiler for a subset of the Java programming language.

Small computer based exercises about formal languages.

The instructions will be on the web, including deadlines.

The project is evaluated during the examination week. If you do not pass, it can be evaluated during the following examination week.
A programming project where you implement a compiler for a subset of the Java programming language.

Small computer based exercises about formal languages.

The instructions will be on the web, including deadlines.

The project is evaluated during the examination week. If you do not pass, it can be evaluated during the following examination week.
A programming project where you implement a compiler for a subset of the Java programming language.

Small computer based exercises about formal languages.

The instructions will be on the web, *including deadlines*.

The project is evaluated during the examination week. If you do not pass, it can be evaluated during the following examination week.
Organization

Theory lectures
On formal languages about
- regular expressions and finite automata
- context free grammars and pushdown automata

Assignments
Short labs to confirm that you understand some theory and the tools we need.

Compiler techniques lectures
- abstract syntax
- types and type checking
- intermediate representations
- code generation and optimizations

Programming project
Organized as a series of laborations with strict deadlines.
Organization

Theory lectures

On formal languages about
- regular expressions and finite automata
- context free grammars and pushdown automata

Assignments

Short labs to confirm that you understand some theory and the tools we need.

Compiler techniques lectures

- abstract syntax
- types and type checking
- intermediate representations
- code generation and optimizations

Programming project

Organized as a series of collaborations with strict deadlines.
**Organization**

**Theory lectures**
- On formal languages about
  - regular expressions and finite automata
  - context free grammars and pushdown automata

**Assignments**
- Short labs to confirm that you understand some theory and the tools we need.

**Compiler techniques lectures**
- abstract syntax
- types and type checking
- intermediate representations
- code generation and optimizations

**Programming project**
- Organized as a series of laborations with strict deadlines.
Organization

Theory lectures
On formal languages about
- regular expressions and finite automata
- context free grammars and pushdown automata

Assignments
Short labs to confirm that you understand some theory and the tools we need.

Compiler techniques lectures
- abstract syntax
- types and type checking
- intermediate representations
- code generation and optimizations

Programming project
Organized as a series of laborations with strict deadlines.
There is plenty of computer languages for plenty of purposes . . .

- *xml* to describe the structure of documents and documents themselves.
- *xquery* to transform *xml* documents.
- *VHDL* to describe circuits.
- *VRML* to describe 3D scenes.
- *C, Java, Haskell* for programming.
There is plenty of computer languages for plenty of purposes . . .

- **xml** to describe the structure of documents and documents themselves.
- **xquery** to transform xml documents.
- **VHDL** to describe circuits.
- **VRML** to describe 3D scenes.
- **C, Java, Haskell** for programming.
Computer Languages

There is plenty of computer languages for plenty of purposes . . .

- *xml* to describe the structure of documents and documents themselves.
- *xquery* to transform xml documents.
- *VHDL* to describe circuits.
- *VRML* to describe 3D scenes.
- *C, Java, Haskell* for programming.
There is plenty of **computer languages** for plenty of purposes . . .

- `xml` to describe the structure of documents and documents themselves.
- `xquery` to transform `xml` documents.
- `VHDL` to describe circuits.
- `VRML` to describe 3D scenes.
- `C, Java, Haskell` for programming.
There is plenty of computer languages for plenty of purposes . . .

- *xml* to describe the structure of documents and documents themselves.
- *xquery* to transform *xml* documents.
- *VHDL* to describe circuits.
- *VRML* to describe 3D scenes.
- *C, Java, Haskell* for programming.
There is plenty of computer languages for plenty of purposes . . .

- `xml` to describe the structure of documents and documents themselves.
- `xquery` to transform `xml` documents.
- `VHDL` to describe circuits.
- `VRML` to describe 3D scenes.
- `C`, `Java`, `Haskell` for programming.
There is plenty of computer languages for plenty of purposes . . .

- *xml* to describe the structure of documents and documents themselves.
- *xquery* to transform *xml* documents.
- *VHDL* to describe *circuits*.
- *VRML* to describe 3D scenes.
- *C, Java, Haskell* for programming.
There is plenty of computer languages for plenty of purposes . . .

- *xml* to describe the structure of documents and documents themselves.
- *xquery* to transform xml documents.
- *VHDL* to describe circuits.
- *VRML* to describe 3D scenes.
- *C, Java, Haskell* for programming.
There is plenty of computer languages for plenty of purposes . . .

- **xml** to describe the structure of documents and documents themselves.
- **xquery** to transform xml documents.
- **VHDL** to describe circuits.
- **VRML** to describe 3D scenes.
- **C, Java, Haskell** for programming.
There is plenty of computer languages for plenty of purposes . . .

- XML to describe the structure of documents and documents themselves.
- XQuery to transform XML documents.
- VHDL to describe circuits.
- VRML to describe 3D scenes.
- C, Java, Haskell for programming.
There is plenty of computer languages for plenty of purposes . . .

- xml to describe the structure of documents and documents themselves.
- xquery to transform xml documents.
- VHDL to describe circuits.
- VRML to describe 3D scenes.
- C, Java, Haskell for programming.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.

...
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- Symbolic manipulations functional languages like Haskell, ML, lisp, scheme or xquery.
- Concurrent languages with communication and synchronization mechanisms like Occam or MPD.
- Embedded languages with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent languages** with communication and synchronization mechanisms like Occam or MPD.
- **Embedded languages** with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.
Different kinds of programs require different kinds of abstractions

- **Symbolic manipulations** functional languages like Haskell, ML, lisp, scheme or xquery.
- **Concurrent** languages with communication and synchronization mechanisms like Occam or MPD.
- **Embedded** languages with interface to hardware like C.

...
All these languages have a lot in common! They all have to be **processed by a program** in order to do what we express in them!

- They are formal languages,
- in most of them we can make definitions,
- in most of them **types** are used to identify meaningful expressions.
Common features

All these languages have a lot in common! They all have to be processed by a program in order to do what we express in them!

- They are formal languages,
- in most of them we can make definitions,
- in most of them types are used to identify meaningful expressions.
All these languages have a lot in common! They all have to be processed by a program in order to do what we express in them!

- They are **formal languages**, 
- in most of them we can make **definitions**, 
- in most of them **types** are used to identify meaningful expressions.
Common features

All these languages have a lot in common! They all have to be processed by a program in order to do what we express in them!

- They are formal languages,
- in most of them we can make definitions,
- in most of them types are used to identify meaningful expressions.
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors

- Compilers
- Interpreters

that have much in common!
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors

- Compilers
- Interpreters

that have much in common!
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors

- Compilers
- Interpreters

They are complex programs that use advanced algorithms and data structures. They show an application of the theory of formal languages we learn how to use tools that generate programs. We learn programming techniques. We learn to do semantic distinctions (for instance different parameter passing mechanisms).
In the course we study a compiler for an imperative programming language.

Language processors
There are two kinds of language processors
- Compilers
- Interpreters

that have much in common!
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors
There are two kinds of language processors
- Compilers
- Interpreters

That have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors
- Compilers
- Interpreters

that have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors
There are two kinds of language processors
- Compilers
- Interpreters
  that have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors

- Compilers
- Interpreters

that have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors
- Compilers
- Interpreters

that have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
Introduction

Regular Expressions

Why a compiler?

In the course we study a compiler for an imperative programming language.

Language processors

There are two kinds of language processors

- Compilers
- Interpreters

that have much in common!

- They are complex programs!
- They use advanced algorithms and data structures.
- They show an application of the theory of formal languages, we learn how to use tools that generate programs.
- We learn programming techniques.
- We learn to do semantic distinctions (for instance different parameter passing mechanisms!)
A compiler is a large, complex program.

- The design and implementation of a compiler is a substantial exercise in software engineering.

A good compiler contains a microcosmos of computer science.

- Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.

Most software is compiled. Compiler construction has given rise to tools for automatic programming that can be used for many purposes. In constructing a compiler you will get to use these tools (and hopefully you will find use for them in other areas!)

**Why a compiler?** Cooper & Torczon. *Engineering a Compiler.* (Elsevier)
Why a compiler? Cooper & Torczon. Engineering a Compiler. (Elsevier)

• A compiler is a large, complex program.
  • The design and implementation of a compiler is a substantial exercise in software engineering.
• A good compiler contains a microcosmos of computer science.
  • Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.
• Most software is compiled. Compiler construction has given rise to tools for automatic programming that can be used for many purposes. In constructing a compiler you will get to use these tools (and hopefully you will find use for them in other areas!)
Why a compiler? Cooper & Torczon. Engineering a Compiler. (Elsevier)

- A compiler is a large, complex program.
  - The design and implementation of a compiler is a substantial exercise in software engineering.

- A good compiler contains a microcosmos of computer science.
  - Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.

- Most software is compiled. Compiler construction has given rise to tools for automatic programming that can be used for many purposes. In constructing a compiler you will get to use these tools (and hopefully you will find use for them in other areas!)
A compiler is a large, complex program.

- *The design and implementation of a compiler is a substantial exercise in software engineering.*

A good compiler contains a microcosmos of computer science.

- *Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.*

Most software is compiled. Compiler construction has given rise to tools for automatic programming that can be used for many purposes. In constructing a compiler you will get to use these tools (and hopefully you will find use for them in other areas!)
A compiler is a large, complex program.

- *The design and implementation of a compiler is a substantial exercise in software engineering.*

A good compiler contains a microcosmos of computer science.

- *Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.*

Most software is compiled. Compiler construction has given rise to tools for automatic programming that can be used for many purposes. In constructing a compiler you will get to use these tools (and hopefully you will find use for them in other areas!)
Why a programming language?

We chose a programming language as running example of a computer language because you are familiar with it and because it illustrates a lot of concepts.

- definitions and scope
- variables and arguments
- types
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages

- lexer generators
- parser generators

Provides you with new **programming techniques and data structures** useful in processing computer languages

- design patterns *component, visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages
- lexer generators
- parser generators

Provides you with new **programming techniques and datastructures** useful in processing computer languages
- design patterns *component, visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages
- lexer generators
- parser generators

Provides you with new **programming techniques and datastructures** useful in processing computer languages
- design patterns *component, visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages
- lexer generators
- parser generators

Provides you with new **programming techniques and datastructures** useful in processing computer languages
- design patterns *component*, *visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages
- lexer generators
- parser generators

Provides you with new **programming techniques and datastructures** useful in processing computer languages
- design patterns *component, visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
Some side effects of the course.

Provides you with **software tools** to describe and implement computer languages
- lexer generators
- parser generators

Provides you with new **programming techniques and datastructures** useful in processing computer languages
- design patterns *component, visitor*
- front end/ back end, abstract machines
- abstract syntax
- environments
You will write a compiler for minijava, a small programming language presented in the course book.

- It is graded, it is the main contribution to the grade of the course!
- It is divided in parts. You will get instructions and deadlines for each part.
- You will get help to get started but you will have to work on your own.
- There will be time for extra questions/supervision during consultation hours, to be announced for each lab.
Programming Project

- You will write a compiler for *minijava*, a small programming language presented in the course book.
- It is graded, it is the main contribution to the grade of the course!
- It is divided in parts. You will get instructions and deadlines for each part.
- You will get help to get started but you will have to work on your own.
- There will be time for extra questions/supervision during consultation hours, to be announced for each lab.
You will write a compiler for minijava, a small programming language presented in the course book.

It is graded, it is the main contribution to the grade of the course!

It is divided in parts. You will get instructions and deadlines for each part.

You will get help to get started but you will have to work on your own.

There will be time for extra questions/supervision during consultation hours, to be announced for each lab.
You will write a compiler for minijava, a small programming language presented in the course book.

It is graded, it is the main contribution to the grade of the course!

It is divided in parts. You will get instructions and deadlines for each part.

You will get help to get started but you will have to work on your own.

There will be time for extra questions/supervision during consultation hours, to be announced for each lab.
You will write a compiler for minijava, a small programming language presented in the course book.

It is graded, it is the main contribution to the grade of the course!

It is divided in parts. You will get instructions and deadlines for each part.

You will get help to get started but you will have to work on your own.

There will be time for extra questions/supervision during consultation hours, to be announced for each lab.
Overview of a compiler

- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!
Overview of a compiler

- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!
Overview of a compiler

- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!
Overview of a compiler

- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!
Overview of a compiler

- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!
Programs are analysed and translated to an intermediate representation IR, a form of abstract machine.

- IR is useful for many things:
  - Detect and remove underflow and overflow
  - Simplify the code to gain efficiency
  - To analyse the code for certain properties
- It also helps to think and understand the different tasks!
Programs are analysed and translated to an intermediate representation IR, a form of abstract machine.

*IR is useful for many things:*
  - detect some errors (indexing out of range)
  - reorganize the code (to gain efficiency)
  - to analyze the code (to assign registers)

*It also helps to think and understand the different tasks!*
Programs are analysed and translated to an intermediate representation **IR**, a form of abstract machine.

**IR** is useful for many things:

- detect some errors (indexing out of range)
- reorganize the code (to gain efficiency)
- to analyze the code (to assign registers)

It also helps to think and understand the different tasks!
Programs are analysed and translated to an intermediate representation IR, a form of abstract machine.

IR is useful for many things:
- detect some errors (indexing out of range)
- reorganize the code (to gain efficiency)
- to analyze the code (to assign registers)

It also helps to think and understand the different tasks!
Programs are analysed and translated to an intermediate representation IR, a form of abstract machine.

**IR** is useful for many things:
- detect some errors (indexing out of range)
- reorganize the code (to gain efficiency)
- to analyze the code (to assign registers)

It also helps to think and understand the different tasks!
Programs are analysed and translated to an intermediate representation IR, a form of abstract machine.

IR is useful for many things:
- detect some errors (indexing out of range)
- reorganize the code (to gain efficiency)
- to analyze the code (to assign registers)

It also helps to think and understand the different tasks!
Programs are analysed and translated to an intermediate representation **IR**, a form of abstract machine.

**IR** is useful for many things:
- detect some errors (indexing out of range)
- reorganize the code (to gain efficiency)
- to analyze the code (to assign registers)

It also helps to think and understand the different tasks!
The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the lexemes of the language.

The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.

This tree is analyzed by the type checker and is then used to generate the intermediate representation.
The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the lexemes of the language.

The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.

This tree is analyzed by the type checker and is then used to generate the intermediate representation.
The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the lexemes of the language.

The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.

This tree is analyzed by the type checker and is then used to generate the intermediate representation.
The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the lexemes of the language.

The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.

This tree is analyzed by the type checker and is then used to generate the intermediate representation.
The back end is also structured in phases!

source code → Front End → IR → Back End → machine code


errors
The back end is also structured in phases!
The back end is also structured in phases!

source code → Front End → IR → Back End → machine code

- The back end is also structured in phases!


errors
Describing a language

What are the phrases of the language? (Syntax)

Example
In English some sentences have the form

\[ \text{<noun phrase> <verb phrase>} \]

where a \text{<noun phrase>} can be

- the \text{<noun>}
- a \text{<noun>}
- \text{name}

What do phrases mean? (Semantics)

Example
In Java the meaning of a statement like

\[ \text{if } \text{<exp>} \text{<stm1>} \text{<stm2>} \]

is given by explaining what happens when it is executed:

When the value of \text{<exp>} is true, \text{<stm1>} is executed, otherwise \text{<stm2>} is executed.
Describing a language

What are the phrases of the language? (Syntax)

Example
In English some sentences have the form

\[ \text{<noun phrase>} \text{<verb phrase>} \]

where a <noun phrase> can be

- the <noun>
- a <noun>
- <name>

What do phrases mean? (Semantics)

Example
In Java the meaning of a statement like

\[ \text{if } \text{<exp>} \text{<stm1>} \text{<stm2>} \]

is given by explaining what happens when it is executed:

When the value of <exp> is true <stm1> is executed, otherwise <stm2> is executed.
Describing a language

What are the **phrases** of the language? **(Syntax)**

Example

In English some sentences have the form

```
<noun_phrase><verb_phrase>
```

where a **<noun_phrase>** can be

- the **<noun>**
- a **<noun>**
- **<name>**

What do phrases **mean?** **(Semantics)**

Example

In Java the meaning of a statement like

```
if <exp><stm1><stm2>
```

is given by explaining what happens when it is executed:

```
When the value of <exp> is true <stm1> is executed, otherwise <stm2> is executed.
```
What are the phrases of the language? (Syntax)

Example
In English some sentences have the form

<noun phrase><verb phrase>

where a <noun phrase> can be

the <noun>
a <noun>
<name>

What do phrases mean? (Semantics)

Example
In Java the meaning of a statement like

```java
if <exp><stm1><stm2>
```

is given by explaining what happens when it is executed:

*When the value of <exp> is true <stm1> is executed, otherwise <stm2> is executed.*
Describing a language

What are the **phrases** of the language? *(Syntax)*

Example

In English some sentences have the form

\[<\text{noun phrase}><\text{verb phrase}>\]

where a **<noun phrase>** can be

- the **<noun>**
- a **<noun>**
- **<name>**

What do phrases **mean?** *(Semantics)*

Example

In Java the meaning of a statement like

\[
\text{if } <\text{exp}><\text{stm1}><\text{stm2}>
\]

is given by explaining what happens when it is executed:

*When the value of <exp> is true <stm1> is executed, otherwise <stm2> is executed.*
Syntax

Alphabets
- Phrases are formed using words.
- Words are formed using characters.

Languages
In the context of our course we will deal with formal languages:

*Sets of strings over some alphabet described by certain rules*
Syntax

Alphabets
- Phrases are formed using **words**.
- Words are formed using **characters**.

Languages
In the context of our course we will deal with **formal languages**:

*Sets of strings over some alphabet described by certain rules*
Syntax

Alphabets
- Phrases are formed using words.
- Words are formed using characters.

Languages
In the context of our course we will deal with formal languages:

Sets of strings over some alphabet described by certain rules
Syntax

- There are different kinds of rules to describe languages.
- According to what kind of rules we use the languages have certain structure and properties.
- Typically languages of words have a simpler structure than languages of phrases.
- Regular expressions are used to describe words of programming languages.
  - Easy to understand, useful in many contexts, software tool support.
Introduction

Regular Expressions

Syntax

- There are different kinds of rules to describe languages.
- According to what kind of rules we use the languages have certain structure and properties.
- Typically languages of words have a simpler structure than languages of phrases.
- Regular expressions are used to describe words of programming languages.
  - Easy to understand, useful in many contexts, software tool support.
There are different kinds of rules to describe languages.

According to what kind of rules we use the languages have certain structure and properties.

Typically languages of words have a simpler structure than languages of phrases.

Regular expressions are used to describe words of programming languages.

Easy to understand, useful in many contexts, software tool support.
There are different kinds of rules to describe languages.

According to what kind of rules we use the languages have certain structure and properties.

Typically languages of words have a simpler structure than languages of phrases.

Regular expressions are used to describe words of programming languages.

Easy to understand, useful in many contexts, software tool support.
Introduction

Regular Expressions

Syntax

There are different kinds of rules to describe languages.

According to what kind of rules we use the languages have certain structure and properties.

Typically languages of words have a simpler structure than languages of phrases.

Regular expressions are used to describe words of programming languages.

- Easy to understand, useful in many contexts, software tool support.
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

- $\epsilon$ is a RE denoting the set with the empty string as only element.
- If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
- If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then $r|s$ is a RE denoting $L(r) \cup L(s)$.
- $rs$ is a RE denoting $\{xy \mid x \in L(r) \text{ and } y \in L(s)\}$.
- $r^*$ is a RE denoting $L(r)^*$. 
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
3. If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then:
   - $(r)$ is a RE denoting $L(r)$.
   - $r|s$ is a RE denoting $L(r) \cup L(s)$.
   - $rs$ is a RE denoting $\{xy | x \in L(r) \text{ and } y \in L(s)\}$.
   - $r^*$ is a RE denoting $L(r)^*$. 
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
3. If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then
   - $r^*$ is a RE denoting $L(r)^*$.
   - $r|s$ is a RE denoting $L(r) \cup L(s)$.
   - $rs$ is a RE denoting $\{xy | x \in L(r) \text{ and } y \in L(s)\}$. 
A regular expression \( r \) over an alphabet \( \Sigma \) describes a set of strings \( L(r) \).

1. \( \epsilon \) is a RE denoting the set with the empty string as only element.
2. If \( a \in \Sigma \) then \( a \) is a RE denoting the set \( \{a\} \).
3. If \( r \) and \( s \) are RE denoting \( L(r) \) and \( L(s) \) respectively, then
   - \( (r) \) is a RE denoting \( L(r) \)
   - \( r | s \) is a RE denoting \( L(r) \cup L(s) \)
   - \( rs \) is a RE denoting \( \{xy | x \in L(r) \text{ and } y \in L(s)\} \)
   - \( r^* \) is a RE denoting \( L(r)^* \)
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. if $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$
3. if $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then
   - $(r)$ is a RE denoting $L(r)$
   - $r|s$ is a RE denoting $L(r) \cup L(s)$
   - $rs$ is a RE denoting $\{xy|x \in L(r) \text{ and } y \in L(s)\}$
   - $r^*$ is a RE denoting $L(r)^*$
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
3. If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then
   - $(r)$ is a RE denoting $L(r)$.
   - $r|s$ is a RE denoting $L(r) \cup L(s)$.
   - $rs$ is a RE denoting $\{xy | x \in L(r) \text{ and } y \in L(s)\}$.
   - $r^*$ is a RE denoting $L(r)^*$. 
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
3. If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then:
   - $(r)$ is a RE denoting $L(r)$
   - $r|s$ is a RE denoting $L(r) \cup L(s)$
   - $rs$ is a RE denoting $\{xy | x \in L(r) \text{ and } y \in L(s)\}$
   - $r^*$ is a RE denoting $L(r)^*$
A regular expression $r$ over an alphabet $\Sigma$ describes a set of strings $L(r)$.

1. $\epsilon$ is a RE denoting the set with the empty string as only element.
2. If $a \in \Sigma$ then $a$ is a RE denoting the set $\{a\}$.
3. If $r$ and $s$ are RE denoting $L(r)$ and $L(s)$ respectively, then
   - $(r)$ is a RE denoting $L(r)$
   - $r|s$ is a RE denoting $L(r) \cup L(s)$
   - $rs$ is a RE denoting $\{xy | x \in L(r) \text{ and } y \in L(s)\}$
   - $r^*$ is a RE denoting $L(r)^*$
We used two operations on sets when introducing regular expressions.

**Union**

\[ A \cup B = \{ x \mid x \in A \text{ or } x \in B \} \]

**Example**

\{0, 2, 4, 6, 8, 10\} \cup \{1, 3, 5, 7, 9, 10\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}

**Kleene closure**

\[ A^* = \epsilon \cup A \cup \{ xy \mid x, y \in A \} \cup \{ xyz \mid x, y, z \in A \} \cup \ldots \]

**Example**

\{a\}^* = \{"", a, aa, aaa, aaaa, aaaaa, \ldots \}
Recall set operations

We used two operations on sets when introducing regular expressions.

### Union

\[ A \cup B = \{ x \mid x \in A \text{ or } x \in B \} \]

**Example**

\{0, 2, 4, 6, 8, 10\} \cup \{1, 3, 5, 7, 9, 10\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}

### Kleene closure

\[ A^* = \epsilon \cup A \cup \{ xy \mid x, y \in A \} \cup \{ xyz \mid x, y, z \in A \} \cup \ldots \]

**Example**

\{a\}^* = \{"", a, aa, aaa, aaaa, aaaaa, \ldots \}
We used two operations on sets when introducing regular expressions.

**Union**

\[ A \cup B = \{ x | x \in A \text{ or } x \in B \} \]

**Example**

\[ \{0, 2, 4, 6, 8, 10\} \cup \{1, 3, 5, 7, 9, 10\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

**Kleene closure**

\[ A^* = \epsilon \cup A \cup \{ xy | x, y \in A \} \cup \{ xyz | x, y, z \in A \} \cup \ldots \]

**Example**

\[ \{a\}^* = \{",", a, aa, aaa, aaaa, aaaaa, \ldots \} \]
Recall set operations

We used two operations on sets when introducing regular expressions.

**Union**

\[ A \cup B = \{ x | x \in A \text{ or } x \in B \} \]

**Example**

\{0, 2, 4, 6, 8, 10\} \cup \{1, 3, 5, 7, 9, 10\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}

**Kleene closure**

\[ A^* = \epsilon \cup A \cup \{ xy | x, y \in A \} \cup \{ xyz | x, y, z \in A \} \cup \ldots \]

**Example**

\{a\}^* = \{"", a, aa, aaa, aaaa, aaaaa, \ldots \}
Recall set operations

We used two operations on sets when introducing regular expressions.

**Union**

\[ A \cup B = \{ x | x \in A \text{ or } x \in B \} \]

**Example**

\{0, 2, 4, 6, 8, 10\} \cup \{1, 3, 5, 7, 9, 10\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}

**Kleene closure**

\[ A^* = \varepsilon \cup A \cup \{ xy | x, y \in A \} \cup \{ xyz | x, y, z \in A \} \cup \ldots \]

**Example**

\{a\}^* = \{””, a, aa, aaa, aaaa, aaaaa, …\}
The set of strings that begin and end with an $a$ and contain at least one $b$:

$$a(a|b)^*b(a|b)^*a$$

Example

$aba$
$aba$, $abaa$, $abba$
$aabaa$, $abbaa$, $aabba$, $ababa$ ...

We omit many parenthesis by following precedence conventions:
- * has highest precedence
- then comes concatenation
- and then union
The set of strings that begin and end with an $a$ and contain at least one $b$:

$$a(a|b)^*b(a|b)^*a$$

Example

$aba$

$aaba$, $abaa$, $abba$

$aabaa$, $abbaa$, $aabba$, $ababa$ ...

...

We omit many parenthesis by following precedence conventions:

- $*$ has highest precedence
- then comes concatenation
- and then union
The set of strings that begin and end with an $a$ and contain at least one $b$:

$$a(a|b)^*b(a|b)^*a$$

Example

- $aba$
- $aaba, abaa, abba$
- $aabaa, abbaa, aabba, ababa$ ...
- ...

We omit many parenthesis by following precedence conventions:

- * has highest precedence
- then comes concatenation
- and then union
Examples & Abbreviations

The set of integer literals

$$0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

Example

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
10, 11, 12, …, 20, 21, …, 99
100, 101, 102 …
…

- We use $[]$ for either:
  $[123456789]$
- We use $-$ for a range in an ordered part of the alphabet:
  $[1 – 9]$

$$0|[1 – 9][0 – 9]^*$$
The set of integer literals

0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*

Example

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
10, 11, 12, ..., 20, 21, ..., 99
100, 101, 102 ...
...

- We use [] for either:
  [123456789]
- We use − for a range in an ordered part of the alphabet: [1 − 9]

0|[1 − 9][0 − 9]*
The set of integer literals

\[ 0(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \]

Example

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
10, 11, 12, \ldots, 20, 21, \ldots, 99
100, 101, 102 \ldots
\ldots

- We use \([\) for either:
  \[123456789]\]
- We use \(--\) for a range in an ordered part of the alphabet:
  \[1 – 9]\]

\[0[1 – 9][0 – 9]^*\]
Identifiers in a little programming language are *words of any length* formed using the characters of the Latin alphabet.

\[ [a - zA - Z][a - zA - Z]^* \]

**Example**

- \( a, b, c, \ldots, A, B, C, \ldots \)
- \( aa, ab, ac, aZ, \ldots \)
- \( \text{the, myX, Int,} \ldots \)
- \( \ldots \)

We use \( r^+ \) instead of \( rr^* \)

\[ [a - zA - Z]^+ \]
Identifiers in a little programming language are words of any length formed using the characters of the latin alphabet.

\[ [a - zA - Z][a - zA - Z]^* \]

Example

- a, b, c, ..., A, B, C, ...
- aa, ab, ac, aZ, ...
- the, myX, Int, ...
- ...

We use \( r^+ \) instead of \( rr^* \)

\[ [a - zA - Z]^+ \]
Identifiers in a little programming language are words of any length formed using the characters of the Latin alphabet.

\[ [a - zA - Z][a - zA - Z]^* \]

Example

\begin{itemize}
  \item \textit{a, b, c, \ldots}, \textit{A, B, C, \ldots}
  \item \textit{aa, ab, ac, aZ, \ldots}
  \item \textit{the, myX, Int, \ldots}
  \item \ldots
\end{itemize}

We use \( r^+ \) instead of \( rr^* \)

\[ [a - zA - Z]^+ \]
Non regular languages

Not all sets of strings are regular!

Example
Given the alphabet $\Sigma = \{a, b\}$, the language $\{a^n b^n | n \geq 0\}$ is not regular.

*It can be proved mathematically, but we will not do it*

However, for any $m \geq 0$, the language $\{a^n b^n | 0 \leq n \leq m\}$ is regular.

Example
Given the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (, )\}$, the set of wellformed arithmetical expressions is not regular.

We need recursion in order to allow for subexpressions and balanced parenthesis.
Not all sets of strings are regular!

Example
Given the alphabet \( \Sigma = \{a, b\} \), the language \( \{a^n b^n \mid n \geq 0\} \) is not regular.

*It can be proved mathematically, but we will not do it*

However, for any \( m \geq 0 \), the language \( \{a^n b^n \mid 0 \leq n \leq m\} \) is regular.

Example
Given the alphabet \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (, )\} \), the set of wellformed arithmetical expressions is not regular.

We need recursion in order to allow for subexpressions and balanced partenthesis.
Non regular languages

Not all sets of strings are regular!

Example
Given the alphabet $\Sigma = \{a, b\}$, the language $\{a^n b^n | n \geq 0\}$ is not regular.

*It can be proved mathematically, but we will not do it*

However, for any $m \geq 0$, the language $\{a^n b^n | 0 \leq n \leq m\}$ is regular.

Example
Given the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (, )\}$, the set of wellformed arithmetical expressions is not regular.

We need recursion in order to allow for subexpressions and balanced partentheses.
Non regular languages

Not all sets of strings are regular!

**Example**

Given the alphabet $\Sigma = \{a, b\}$, the language $\{a^n b^n | n \geq 0\}$ is not regular.

*It can be proved mathematically, but we will not do it*

However, for any $m \geq 0$, the language $\{a^n b^n | 0 \leq n \leq m\}$ is regular.

**Example**

Given the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (, )\}$, the set of wellformed arithmetical expressions is not regular.

We need recursion in order to allow for subexpressions and balanced parentheses.
Introduction

Regular Expressions

Non regular languages

Not all sets of strings are regular!

Example
Given the alphabet $\Sigma = \{a, b\}$, the language $\{a^n b^n | n \geq 0\}$ is not regular.

*It can be proved mathematically, but we will not do it*

However, for any $m \geq 0$, the language $\{a^n b^n | 0 \leq n \leq m\}$ is regular.

Example
Given the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (, )\}$, the set of wellformed arithmetical expressions is not regular.

We need recursion in order to allow for subexpressions and balanced partenthesis.
Regular expressions on the web

There is a lot of material on the web, both in the form of lecture notes, slides and books. I will not link from the web page, but I will put some links on the slides that you can check.

**A book on compilers**

N. Wirth. *Compiler Construction*.
http://www.oberon.ethz.ch/WirthPubl/CBEAll.pdf

**Lecture notes for a compiler course**

http://lambda.uta.edu/cse5317/notes/

**A similar course at Luleå**

Where you can find slides.
http://www.sm.luth.se/csee/courses/smd/163/
Scanners

A *scanner* is a program that inspects a sequence of characters trying to identify words of a language. They can be used for many different purposes.

**Example**

In a compiler a scanner (or lexical analyzer) is used to begin with the analysis (understanding) of the source code:

- Read the sequence of characters and produce a sequence of tokens
- Facilitates the analysis of the phrases of the program
- Interrupt the compilation process in case of some lexicographic error, including reporting an error message.
A *scanner* is a program that inspects a sequence of characters trying to identify words of a language. They can be used for many different purposes.

**Example**

In a compiler a scanner (or lexical analyzer) is used to begin with the analysis (understanding) of the source code:

- Read the sequence of characters and produce a sequence of tokens
- Facilitates the analysis of the phrases of the program
- Interrupt the compilation process in case of some lexicographic error, including reporting an error message.
A *scanner* is a program that inspects a sequence of characters trying to identify words of a language. They can be used for many different purposes.

**Example**

In a compiler a scanner (or lexical analyzer) is used to begin with the analysis (understanding) of the source code:

- Read the sequence of characters and produce a sequence of tokens
  - Facilitates the analysis of the phrases of the program!
- Interrupt the compilation process in case of some lexicographic error, including reporting an error message.
A scanner is a program that inspects a sequence of characters trying to identify words of a language. They can be used for many different purposes.

Example

In a compiler a scanner (or lexical analyzer) is used to begin with the analysis (understanding) of the source code:

- Read the sequence of characters and produce a sequence of tokens
  - Facilitates the analysis of the phrases of the program!
- Interrupt the compilation process in case of some lexicographic error, including reporting an error message.
A scanner is a program that inspects a sequence of characters trying to identify words of a language. They can be used for many different purposes.

**Example**

In a compiler a scanner (or lexical analyzer) is used to begin with the analysis (understanding) of the source code:

- Read the sequence of characters and produce a sequence of tokens
  - Facilitates the analysis of the phrases of the program!
- Interrupt the compilation process in case of some lexicographic error, including reporting an error message.
Scanners are tedious programs to write, with many cases to take care of, very error prone!

Some math that we will discuss in the second lecture has resulted in programs that generate scanners from a regular expression.

We will use , written in Java and generating a Java program.
Scanners are tedious programs to write, with many cases to take care of, very error prone!

Some math that we will discuss in the second lecture has resulted in programs that generate scanners from a regular expression.

We will use , written in Java and generating a Java program.
Scanners are tedious programs to write, with many cases to take care of, very error prone!

Some math that we will discuss in the second lecture has resulted in programs that generate scanners from a regular expression.

We will use jFlex, written in Java and generating a Java program.
Scanners are tedious programs to write, with many cases to take care of, very error prone!

Some math that we will discuss in the second lecture has resulted in programs that generate scanners from a regular expression.

We will use JFlex, written in Java and generating a Java program.
Regular expressions, directives, java code.

```java
// code outside MyName
%%%

%unicode
%int
%class MyName
%function next
{%
// code inside MyName
%}
%%%

"veronica gaspes"
   {System.out.println(yytext());}
.\n{n{`
```
Regular expressions, directives, java code.

JFlex the source!

Scanner in Java, compile it!

Run the scanner on a file of text!

```java
// code outside MyName
%%%
%unicode
%int
%class MyName
%function next
{%
// code inside MyName
%
%%%
"veronica gaspes"
   {System.out.println(yytext());}
. | \n{}
```
JFlex source

Regular expressions, directives, java code.

JFlex the source!

Scanner in Java, compile it!

Run the scanner on a file of text!

// code outside MyName
%
%

%unicode
%int
%class MyName
%function next
%
// code inside MyName
%
%
"veronica gaspes"
    {System.out.println(yytext());}
  .|\n{}
Directives and conventions

- **%line** allows you to use the variable `yyline` that is automatically incremented on every line change.
- **%column** allows you to use the variable `yycolumn` that is automatically incremented and reinitialized.
- **%implements** InterfaceName lets the generated java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.

- Always go for the longest sequence that matches an expression.
- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.
Directives and conventions

- **%line** allows you to use the variable *yyline* that is automatically incremented on every line change.
- **%column** allows you to use the variable *yycolumn* that is automatically incremented and reinitialized.
- **%implements InterfaceName** lets the generated Java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.

- Always go for the longest sequence that matches an expression.
- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.
Directives and conventions

- **%line** allows you to use the variable `yyline` that is automatically incremented on every line change.
- **%column** allows you to use the variable `yycolumn` that is automatically incremented and reinitialized.
- **%implements**
  InterfaceName lets the generated java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.
- Allways go for the longest sequence that matches an expression.
- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.
Directives and conventions

- `%line` allows you to use the variable `yyline` that is automatically incremented on every line change.
- `%column` allows you to use the variable `yycolumn` that is automatically incremented and reinitialized.
- `%implements` InterfaceName lets the generated java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.

- Allways go for the longest sequence that matches an expression.
- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.
Directives and conventions

- **%line** allows you to use the variable `yyline` that is automatically incremented on every line change.

- **%column** allows you to use the variable `yycolumn` that is automatically incremented and reinitialized.

- **%implements** InterfaceName lets the generated java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.

- Always go for the longest sequence that matches an expression.

- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.
Directives and conventions

- **%line** allows you to use the variable `yyline` that is automatically incremented on every line change.
- **%column** allows you to use the variable `yycolumn` that is automatically incremented and reinitialized.
- **%implements** `InterfaceName` lets the generated Java class implement the interface.

When scanning the input sequence of characters, there might be clashes between some of the regular expressions.

- Always go for the longest sequence that matches an expression.
- The order of the rules indicates precedence, if a word is described by more than one RE the first one will be chosen.