**Organizing**

- Master or Bachelor in the 6th Semester with 5 ECTS
- Prerequisites
  - Informatik 1 & 2
  - Theoretische Informatik
  - Technische Informatik
  - Grundlegende Algorithmen
- Delve deeper with
  - Virtual Machines
  - Programmoptimization
  - Programming Languages
  - Praktikum Compilerbau
  - Hauptseminars

**Materials:**

- TTT-based lecture recordings
- the slides
- Related literature list online
- Tools for visualization of virtual machines
- Tools for generating components of Compilers

---

**Organizing**

**Dates:**

- Lecture: Mo. 14:15-15:45
- Tutorial: You can vote on two dates via moodle

**Exam:**

- One Exam in the summer, none in the winter
- Exam managed via TUM-online
- Successful (50% credits) tutorial exercises earns 0.3 bonus
Preliminary content

- Basics in regular expressions and automata
- Specification and implementation of scanners
- Reduced context free grammars and pushdown automata
- Bottom-Up Syntaxanalysis
- Attribute systems
- Typechecking
- Codegeneration for stack machines
- Register assignment
- Basic Optimization

Interpreter

Pro:
No precomputation on program text necessary
⇒ no/small Startup-time

Con:
Program components are analyzed multiple times during the execution
⇒ longer runtime

Concept of a Compiler:

Two Phases:
1. Translating the program text into a machine code
2. Executing the machine code on the input
A precomputation on the program allows
- a more sophisticated variable management
- discovery and implementation of global optimizations

Disadvantage
The translation costs time

Advantage
The execution of the program becomes more efficient
⇒ payoff for more sophisticated or multiply running programs.

The Analysis-Phase is divided in several parts:

Lexicographic Analysis:
Partitioning in tokens

(annotated) Syntax tree
The **Analysis-Phase** is divided in several parts:

- **Scanner**
  - Token-Stream
  - lexicographic Analysis:
    - Partitioning in tokens

- **Parser**
  - syntactic Analysis:
    - Detecting hierarchical structure

- **Syntax tree**

(annotated) Syntax tree

**Topic:**

**Lexical Analysis**
A Token is a sequence of characters which together form a unit.

- Tokens are subsumed in classes. For example:
  - Names (Identifiers) e.g. `xyz, pi, ...`
  - Constants e.g. `42, 3.14, "abc" ...`
  - Operators e.g. `+`, ...
  - reserved terms e.g. `if, int, ...`

Classified tokens allow for further pre-processing:

- Dropping irrelevant fragments e.g. Spacing, Comments, ...
- Separating Pragmas, i.e. directives for the compiler, which are not directly part of the program, like `include-Statements`;
- Replacing of Tokens of particular classes with their meaning / internal representation, e.g.
  - Constants;
  - Names: typically managed centrally in a `Symbol-table`, evt. compared to reserved terms (if not already done by the scanner) and possibly replaced with an index.
The lexical Analysis

Discussion:

- Scanner and Siever are often combined into a single component, mostly by providing appropriate callback actions in the event that the scanner detects a token.
- Scanners are mostly not written manually, but generated from a specification.

Advantages

Productivity: The component can be produced more rapidly.
Correctness: The component implements (provably) the specification.
Efficiency: The generator can provide the produced component with very efficient algorithms.

Disadvantages

- Specification is just another form of programming — admittedly possibly simpler.
- Generation instead of implementatation pays off for Routine-tasks only
  ... and is only good for problems, that are well understood.

The lexical Analysis - Generating:

Discussion:

- Scanner and Siever are often combined into a single component, mostly by providing appropriate callback actions in the event that the scanner detects a token.
- Scanners are mostly not written manually, but generated from a specification.

Advantages

Productivity: The component can be produced more rapidly.
Correctness: The component implements (provably) the specification.
Efficiency: The generator can provide the produced component with very efficient algorithms.

Disadvantages

- Specification is just another form of programming — admittedly possibly simpler.
- Generation instead of implementatation pays off for Routine-tasks only
  ... and is only good for problems, that are well understood.

The lexical Analysis - Generating:

... in our case:

Specification → Generator → Scanner
The lexical Analysis - Generating:

... in our case:

**Specification of Token-classes:** Regular expressions;  
**Generated Implementation:** Finite automata + $X$

### Regular expressions

#### Basics
- Program code is composed from a finite alphabet $\Sigma$ of input characters, e.g. Unicode
- The sets of text fragments of a token class are in general regular.
- Regular languages can be specified by regular expressions.

#### Definition Regular expressions

The set $\mathcal{E}_\Sigma$ of (non-empty) regular expressions is the smallest set $\mathcal{E}$ with:

- $e \in \mathcal{E}$ (a new symbol not from $\Sigma$);
- $a \in \mathcal{E}$ for all $a \in \Sigma$;
- $(e_1, e_2), e_1, e_2 \in \mathcal{E}$ if $e_1, e_2 \in \mathcal{E}$. 

---

Chapter 1: Basics: Regular Expressions

#### Regular expressions

#### Basics
- Program code is composed from a finite alphabet $\Sigma$ of input characters, e.g. Unicode
- The sets of text fragments of a token class are in general regular.
- Regular languages can be specified by regular expressions.
Regular expressions

... Example:

\[(a \cdot b^*) \cdot a\]
\[(a \mid b)\]
\[((a \cdot b) \cdot (a \cdot b))\]

Attention:
- We distinguish between characters \(a, 0, \$, \ldots\) and Meta-symbols \((, \mid, \cdot, \ldots)\).
- To avoid (ugly) parantheses, we make use of Operator-Precedences:

\[^* > \cdot > |\]

and omit ".".
- Real Specification-languages offer additional constructs:

\[e^* \equiv \{\varepsilon \cdot e^*\}\]

and omit "\(e\)."

Keep in mind:
- The operators \((\_)^*, \cup, \cdot\) are interpreted in the context of sets of words:

\[L^{*} = \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\}\]
\[L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}\]

Specifications need Semantics

... Example:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(abab)</td>
<td>{abab}</td>
</tr>
<tr>
<td>(a \mid b)</td>
<td>{a, b}</td>
</tr>
<tr>
<td>(ab^*a)</td>
<td>{ab^n a \mid n \geq 0}</td>
</tr>
</tbody>
</table>

For \(e \in \Sigma^*\), we define the specified language \([e] \subseteq \Sigma^*\) inductively by:

- \([e] = \{e\}\)
- \([a] = \{a\}\)
- \([e^*] = (\{e\})^*\)
- \([e_1 \cup e_2] = [e_1] \cup [e_2]\)
- \([e_1 \cdot e_2] = [e_1] \cdot [e_2]\)
Keep in mind:

- The operators $(\_)^*, \cup, \cdot$ are interpreted in the context of sets of words:

\[
(L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\}
\]

\[
L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}
\]

- Regular expressions are internally represented as annotated ranked trees:

Inner nodes: Operator applications;
Leaves: particular symbols or $\varepsilon$.

Regular expressions

Example: Identifiers in Java:

le = [a-zA-Z_\$]
di = [0-9]
Id = {le} (|{le} | {di})*

Regular expressions

Example: Identifiers in Java:

le = [a-zA-Z_\$]
di = [0-9]
Id = {le} (|{le} | {di})*
Float = (di|s ([.](di)|{di}\.) (di)*((e|E)(\+|\-)? di)*
Regular expressions

Example: Identifiers in Java:
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} {le} | {di})*
Float = {di}+ (\.{di}\{di}\.) {di}+((e|E)(\+|\-)?{di}+)?

Remarks:
- "le" and "di" are token classes.
- Defined Names are enclosed in "{}".
- Symbols are distinguished from Meta-symbols via "\".

Finite automata

Definition
A non-deterministic finite automaton (NFA) is a tuple \( A = (Q, \Sigma, \delta, I, F) \) with:

\[
\begin{align*}
Q & \quad \text{a finite set of states;} \\
\Sigma & \quad \text{a finite alphabet of inputs;} \\
I & \subseteq Q \quad \text{the set of start states;} \\
F & \subseteq Q \quad \text{the set of final states and} \\
\delta & \quad \text{the set of transitions (-relation.)}
\end{align*}
\]

For an NFA, we reckon:

Definition
Given \( \delta : Q \times \Sigma \rightarrow Q \) a function and \(|I| = 1\), then we call \( A \) deterministic (DFA).
Finite automata

- Computations are paths in the graph.
- Accepting computations lead from $I$ to $F$.
- An accepted word is the sequence of labels along an accepting computation ...

In linear time from Regular Expressions to NFAs

Thompson’s Algorithm
Produces $O(n)$ states for regular expressions of length $n$. 