Organizing

- Master or Bachelor in the 8th 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

Zeiten:
- Lecture: Mo. 14:15-15:45
- Tutorial: Tuesday morning and afternoon

Exam
- Exam managed via TUM-online
- Successful tutorial exercises earns 0.3 bonus

Preliminary content

- Basics in regular expressions and automata
- Specification with regular expressions and implementation with automata
- Reduced context free grammars and pushdown automata
- Bottom-Up Syntaxanlysis
- Attribute systems
- Typechecking
- Codegeneration for stack machines
- Register assignment
- Basic Optimization
**Introduction**

**Concept of a Compiler:**
- Program → Compiler → Code
- Code → Machine → Output

**Two Phases:**
1. Translating the program text into a machine code
2. Executing the machine code on the input

**Interpreter**
- Program → Interpreter → Output

**Pro:**
- No precomputation on program text necessary
- Small Startup-time

**Con:**
- Program components are analyzed multiple times during the execution
- Longer runtime

**Compiler**
- A precomputation on the program allows:
  - More sophisticated variable management
  - Discovery and implementation of global optimizations

**Disadvantage**
- 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:

- **Lexical Analysis**: Partitioning in tokens
  - **Scanner**
    - Token-Stream
  - (annotated) Syntax tree

- **Syntax Analysis**: Detecting hierarchical structure
  - **Parser**
    - Syntax tree
Compiler

The Analysis-Phase is divided in several parts:

- Lexicographic Analysis: Partitioning in tokens
- Syntactic Analysis: Detecting hierarchical structure
- Semantic Analysis: Inferring semantic properties

The lexical Analysis

Program code → Scanner → Token-Stream

The lexical Analysis

xyz + 42 → Scanner → xyz + 42

- 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...
The 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, ...

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.

The lexical Analysis

**Classified tokens allow for further pre-processing:**
- **Dropping** irrelevant fragments e.g. Spacing, Comments,...
- **Separating Pragmas** i.e. directives vor 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, e.g. compared to reserved terms (if not already done by the scanner) and possibly replaced with an index.

⇒ Siever

The lexical Analysis - Generating:

**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.
The lexical Analysis - Generating:

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

... in our case:

Specification  ➔  Generator  ➔  Scanner

The lexical Analysis - Generating:

... in our case:

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

Lexikacal Analysis

Kapitel 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 textfragments of a token class is in general regular.
- Regular languages can be specified by regular expressions.

Definition Regular expressions
The set $E_\Sigma$ of (non-empty) regular expressions is the smallest set $E$ with:
- $\epsilon \in E$ ($\epsilon$ a new symbol not from $\Sigma$);
- $a \in E$ for all $a \in \Sigma$;
- $(e_1 \cdot e_2), (e_1 \cdot e_2), e_1^* \in E$ if $e_1, e_2 \in E$.

Regular expressions

... Example:

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

Regular expressions

... Example:

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

Attention:
- We distinguish between characters $a, 0, $,... and Meta-symbols ($, | )$.
- To avoid (ugly) parantheses, we make use of Operator-Precendences:
  $^* > \cdot > |$
- and omit "•".
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 \((, |, )\).
- To avoid (ugly) parantheses, we make use of Operator-Precedences:
  
  \[\ast > \cdot > |\]

and omit “.”
- Real Specifications-languages offer additional constructs:
  
  \[e^2 \equiv (e \mid e)\]
  \[e^+ \equiv (e \cdot e^*)\]

and omit “e”

Beachte:

- The operators \((,), \cup, \cdot\) are interpreted in the context of sets of words:
  
  \[(L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_j \in L\}\]
  
  \[L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}\]

Regular expressions 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^*a \mid n \geq 0})</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
[e] & = \{e\} \\
[a] & = \{a\} \\
[e^*] & = (\{e\})^* \\
[e_1 + e_2] & = e_1 \cup e_2 \\
[e_1 \cdot e_2] & = e_1 \cdot e_2
\end{align*}
\]

Beachte:

- The operators \((,), \cup, \cdot\) are interpreted in the context of sets of words:
  
  \[(L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_j \in L\}\]
  
  \[L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}\]

Regular expressions are internally represented as annotated ranked trees:

- Inner nodes: Operator-applications;
- Leaves: particular symbols or \(\epsilon\).
Regular expressions

Example: Identifiers in Java:

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

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

Finite automata

Example:

Nodes: States;
Edges: Transitions;
Labels: Consumed input;
**Finite automata**

**Definition**

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

- \( Q \) a finite set of states;
- \( \Sigma \) a finite alphabet of inputs;
- \( I \subseteq Q \) the set of start states;
- \( F \subseteq Q \) the set of final states and
- \( \delta \) the set of transitions (-relation)

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 ...

![Diagram](image1)

**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 ...

![Diagram](image2)
Kapitel 3:
Converting Regular expressions in NFAs