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
  - Seminars

Materials:
- TTT-based lecture recordings
- The slides
- Related literature list online (→ Wilhelm/Seidl/Hack Compiler Design)
- Tools for visualization of virtual machines (VAM)
- Tools for generating components of Compilers (JFlex/CUP)

Organizing

Dates:
- Lecture: Mo 14:15-15:45
- Tutorial: Mo 16:00-18:00 and Tue 14:00-16:00 in MI 02.07.014

Exam:
- One Exam in the summer, none in the winter
- Exam managed via TUM-online/campus
- Successful mini project earns 0.3 bonus

Mini Projects:
- A practical implementation, based on a compiler fragment
- Implement a subcomponent:
  - Type system (memory model)
  - Typecasts
  - Type verification
  - Additional Language Features (ellipses, enums, unions)
  - Code generation for Raspberry Pi/ARM
Preliminary content

- Regular expressions and finite automata
- Specification and implementation of scanners
- Reduced context-free grammars and pushdown automata
- Top-Down/Bottom-Up syntax analysis
- Attribute systems
  - Typechecking
  - Codegeneration for register machines
- Register assignment
- Optional: Basic optimization

Interpreter

Program \[\rightarrow\] Interpreter \[\rightarrow\] Output

Con: Program components are analyzed multiple times during execution
  \[\Rightarrow\] longer runtime

Pro: No precomputation on program text necessary
  \[\Rightarrow\] no/small Startup-time

Concept of a Compiler

Program \[\rightarrow\] Compiler \[\rightarrow\] Code

Code \[\rightarrow\] Machine \[\rightarrow\] Output

Two Phases:
- Translating the program text into a machine code
- 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
\(\Rightarrow\) payoff for more sophisticated or multiply running programs.

**general Compiler setup:**

**The Analysis-Phase consists of several subcomponents:**
The **Analysis-Phase** consists of several subcomponents:

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

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

- **Type Checker...**
  - (annotated) Syntax tree
  - semantic Analysis: Inferring semantic properties
A Token is a sequence of characters, which together form a unit.

Tokens are subsumed in classes. For example:

- Names (Identifiers) e.g. \texttt{xyz}, \texttt{pi}, ...
- Constants e.g. 42, \texttt{3.14}, "abc", ...
- Operators e.g. +, ...
- Reserved terms e.g. \texttt{int}, ...

Classified tokens allow for further pre-processing:

- Dropping irrelevant fragments e.g. Spacing, Comments,...
- Collecting Pragmas, i.e. directives for the compiler, which are not directly part of the source language, like OpenMP-Statements;
- Replacing of Tokens of particular classes with their meaning / internal representation, e.g.
  - \textbf{Constants};
  - \textbf{Names}: typically managed centrally in a Symbol-table, may be compared to reserved terms (if not already done by the scanner) and possibly replaced with an index or internal format (\textbf{Name Mangling}).

$\Rightarrow$ Siever
The Lexical Analysis

Discussion:
- Scanners and Sieves 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.

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 is in general regular.
- Regular languages can be specified by regular expressions.

... in our case:

Specification \( \rightarrow \) Generator \( \rightarrow \) Scanner

Specification of Token-classes:

Generated Implementation:

Regular expressions:

Finite automata + \( \times \)
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:
- $\varepsilon \in E$ (a new symbol not from $\Sigma$);
- $a \in E$ for all $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), e^* \in E$ if $e_1, e_2 \in E$.

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, \$, ..., and Meta-symbols $(, |, )$,...
- To avoid (ugly) parantheses, we make use of Operator-Precedences:

.. latex-code-block::

    $>$ $> >$

    and omit “.”

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

.. latex-code-block::

    $>$ $> >$

    and omit “.”
- Real Specification-languages offer additional constructs:

.. latex-code-block::

    $e^*$ $\equiv (e \cdot e)$

    and omit “$*$"
Regular Expressions

Specification needs 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</td>
<td>b</td>
</tr>
<tr>
<td>ab*a</td>
<td>{ab^n a</td>
</tr>
</tbody>
</table>

For \( e \in \mathcal{E}_\Sigma \) we define the specified language \([e] \subset \Sigma^*\) inductively by:

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

Keep in Mind:

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

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

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

**Examples of Regular Expressions**

Keep in Mind:

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

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

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

- Regular expressions are internally represented as annotated ranked trees:

**Inner nodes:** Operator-applications;

**Leaves:** particular symbols or \(\epsilon\).
Regular Expressions

Example: Identifiers in Java:

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

\( Float = \{di\} (\{di\}|\{di\}.)*(e|E)(\+|\-)?\{di\}+ \)

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

Finite Automata

Example:

Nodes: States;
Edges: Transitions;
Labels: Consumed input;
Definition Finite Automata

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 Deterministic Finite Automata

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

Finite Automata

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

Once again, more formally:

- We define the transitive closure $\delta^*$ of $\delta$ as the smallest set $\delta'$ with:
  
  \[
  \begin{align*}
  (p, \epsilon, p) \in \delta' & \quad \text{and} \\
  (p, x, q) \in \delta & \quad \text{if} \quad (p, x, q) \in \delta;
  \end{align*}
  \]

  $\delta^*$ characterizes for two states $p$ and $q$ the words, along each path between them

- The set of all accepting words, i.e. $A$'s accepted language can be described compactly as:

  \[
  L(A) = \{ w \in \Sigma^* | \exists i \in I \exists f \in F : (w, f) \in \delta^* \}
  \]
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

**Berry-Sethi Approach**

**Berry-Sethi Algorithm**

- Produces exactly $n + 1$ states without $\epsilon$-transitions
- and demonstrates $\rightarrow$ *Equality Systems* and $\rightarrow$ *Attribute Grammars*

**Idea:**

- The automaton tracks (conventionally via a marker $\ast$), in the syntax tree of a regular expression, which subexpressions in $\epsilon$ are reachable consuming the rest of input $\mathcal{M}$.

**Thompson’s Algorithm**

- Produces $O(n)$ states for regular expressions of length $n$. 

Lexical Analysis

Chapter 3:

Converting Regular Expressions to NFAs
Berry-Sethi Approach

Glushkov Algorithm
Produces exactly $n + 1$ states without $\epsilon$-transitions and demonstrates $\rightarrow$ Equality Systems and $\rightarrow$ Attribute Grammars

Viktor M. Glushkov

Idea:
The automaton tracks (conceptionally via a marker "\*"), in the syntax tree of a regular expression, which subexpressions in $\epsilon$ are reachable consuming the rest of input $w$. 