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

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)
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:

The Analysis-Phase is divided in several parts:

lexicographic Analysis:
Partitioning in tokens
The **Analysis-Phase** is divided in several parts:

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

```c
int f() {  
    int x, i;  
    x = y + 2*i;  
}
```

```
1hs  
/  
\  
1hs  
/  
\  
decl 
\  
1hs  
/  
\  
add   
\  
1hs  
/  
\  
sub   
\  
1hs  
/  
\  
acc   
\  
(annotated) Syntax tree
```
A Token is a sequence of characters, which together form a unit. Tokens are subsumed in classes. For example:

- Names (Identifiers) e.g. $xyz, p1, ...$
- Constants e.g. $42, 3.14, "abc", ...$
- Operators e.g. $+, ...$
- reserved terms e.g. $if, int, ...$
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, evtl. compared to reserved terms (if not already done by the scanner) and possibly replaced with an index.

⇒ Siever

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. \textit{xyz}, \textit{pi}, ...
  - Constants e.g. \textit{42}, \textit{3.14}, “\textit{abc}”, ...
  - Operators e.g. \textit{+}, ...
  - reserved terms e.g. \textit{if}, \textit{int}, ...

⇒ Siever

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 - Generating:

... in our case:

Specification → Generator → Scanner

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. \texttt{xyz, pi, ...}
  - Constants e.g. \texttt{42, 3.14, "abc", ...}
  - Operators e.g. \texttt{+ ...}
  - reserved terms e.g. \texttt{if, int, ...}

The Lexical Analysis - Generating:

- Specifications

The Lexical Analysis - Generating:

... in our case:

Specification → Generator → Scanner

... in our case:

Specification of Token-classes: Regular expressions;
Generated Implementation: Finite automata + X
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 textfragments of a token class is in general regular.
- Regular languages can be specified by regular expressions.

Definition Regular Expressions

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

- \( \epsilon \in \mathcal{E} \) (\( \epsilon \) a new symbol not from \( \Sigma \));
- \( a \in \mathcal{E} \) for all \( a \in \Sigma \);
- \( (e_1, e_2, e_3) \in \mathcal{E} \) if \( e_1, e_2 \in \mathcal{E} \).

Example:

\[
(a \cdot b^* \cdot a) \\
(a \cdot b) \\
((a \cdot b) \cdot (a \cdot b))
\]
Regular Expressions

... Example:

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

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

\[ " > > | \]

and omit ".".

Regular Expressions

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</td>
<td>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:

\[\begin{align*}
[e] &= \{e\} \\
[\epsilon] &= \{\epsilon\}^* \\
[e_1 e_2] &= [e_1] \cup [e_2] \\
[e_1 e_2] &= [e_1] \cdot [e_2]
\end{align*}\]
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:

\[ (ab|\epsilon)^* \]

**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))*

Lexical Analysis

Chapter 2:  
Basics: Finite Automata

Example: Identifiers in Java:

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

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

Example:

Finite Automata

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

- 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...
**Finite Automata**

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

**Example:**

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

**Finite Automata**

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

- **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 ...
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 ...

Lexical Analysis

Chapter 3:
Converting Regular Expressions to NFAs

Finite Automata

Once again, more formally:

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

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

In linear time from Regular Expressions to NFAs

**Thompson’s Algorithm**

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

Ken Thompson