Title: Petter: Compilerbau (27.04.2017)
Date: Thu Apr 27 14:20:56 CEST 2017
Duration: 90:48 min
Pages: 18

Organizing

- Master or Bachelor in the 6th Semester with 5 ECTS
- Prerequisites
  - Informatik 1 & 2, especially: Java
  - 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)

Dates:
- Lecture: Thursday 14:15-15:45
- Tutorial: Tbd in doodle until Fri. 28th 17:00

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 tutorial in a specific compiler related topic, e.g.
- generating a Java-based parser with CUP/JFlex
- Attribute Grammars with JastAdd
- Attribute Grammars with UUAGC
- The Coco/R Compiler Generator System
- Extensible Grammars with PPG
- coupling with LLVM Codegeneration API
- accessing the LLVM C++ parse tree
- generating a C++ based parser with ANTLR
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

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

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

Concept of a Compiler

![Diagram of the concept of a compiler]

Two Phases:
- Translating the program text into a machine code
- Executing the machine code on the input

Compiler

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 Lexical Analysis

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.
  - Constants;
  - Names: typically managed centrally in a Symbol-table, maybe compared to reserved terms (if not already done by the scanner) and possibly replaced with an index or internal format (⇒ Name Mangling).

⇒ 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**

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.

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

Regular Expressions

... Example:
- $(a \cdot b^*) \cdot a$
- $(a \cdot b)$
- $((a \cdot b) \cdot (a \cdot b))$

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>$a \cdot b$</td>
<td>$a \cdot b$</td>
</tr>
<tr>
<td>$ab$</td>
<td>$ab^{\geq u}$</td>
</tr>
</tbody>
</table>

For $e \in \mathcal{E}$ we define the specified language inductively by:

\[
\begin{align*}
\gamma^* &= \{ \varepsilon \} \\
\gamma \cdot \delta^* &= \gamma \cdot \delta^* \\
\gamma \cup \delta^* &= \gamma \cup \delta^* \\
\gamma \cdot \delta &= \gamma \cdot \delta \\
\end{align*}
\]
Definition Finite Automata

A non-deterministic finite automaton (NFA) is a tuple \( A = (Q, \Sigma, \delta, I, F) \) with:
- \( Q \) is a finite set of states;
- \( \Sigma \) is a finite alphabet of inputs;
- \( I \subseteq Q \) is the set of start states;
- \( F \subseteq Q \) is the set of final states and
- \( \delta \) is the set of transitions (relation).

Once again, more formally:
- We define the transitive closure \( \delta^* \) of \( \delta \) as the smallest set \( \delta^* \) with:
  \[
  \begin{align*}
  (p, r, p) & \in \delta^* \quad \text{and} \\
  (p, r, q, s) & \in \delta^* \quad \text{if} \quad (p, r, q) \in \delta \quad \text{and} \quad (q, s) \in \delta^*.
  \end{align*}
  \]
- \( \delta^* \) 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:
  \[
  \mathcal{L}(A) = \{ w \in \Sigma^* | \exists \ i \in I, f \in F : (i, w, f) \in \delta^* \}.
  \]