Script **generated by TTT**

Title: Petter: Compilerbau (27.04.2017)

Date: Thu Apr 27 14:20:56 CEST 2017

Duration: 90:48 min

Pages: 18

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compiler Construction I

Dr. Michael Petter

SoSe 2017

Organizing ttt.in.tum.de

- 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)

Organizing www2.in.tum.de

Dates:

Lecture: Thursday 14:15-15:45
Tutorial: Tbd in doodle until *Fr. 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

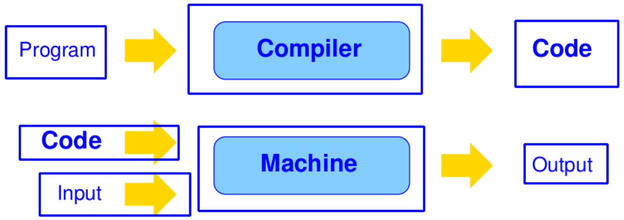
- Regular expressions and finite automata
- Specification and implementation of scanners
- Reduced context free grammars and pushdown automata
- Top-Down/Bottom-Up syntaxanalysis
- Attribute systems
- Typechecking
- Codegeneration for register machines
- Register assignment
- Optional: Basic optimization

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



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

Compiler

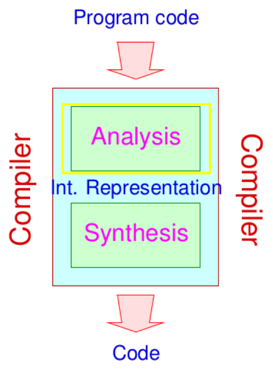
A precomputation on the program allows
- a more sophisticated variable management
- discovery and implementation of global optimizations

Disadvantage
The Translation costs time

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

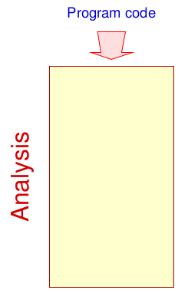
Compiler

general Compiler setup:



Compiler

The Analysis-Phase consists of several subcomponents:



Identifiers
Int Const
Operator

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

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 - Generating:

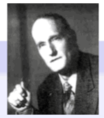
... in our case:

Specification ➡ **Generator** ➡ Scanner

Regular Expressions

Basics

- Program code is composed from a finite alphabet Σ 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.



Stephen Kleene

Definition Regular Expressions

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

- $\epsilon \in \mathcal{E}$ (ϵ a new symbol not from Σ);
- $a \in \mathcal{E}$ for all $a \in \Sigma$;
- $\boxed{(e_1 \mid e_2)}, \boxed{(e_1 \cdot e_2)}, \boxed{e_1{}^*} \in \mathcal{E}$ if $e_1, e_2 \in \mathcal{E}$.

Regular Expressions

... Example:

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

Regular Expressions

Specification needs Semantics

...Example:

Specification	Semantics
$abab$	$\{abab\}$
$a \mid b$	$\{a, b\}$
ab^*a	$\{ab^n a \mid n \geq 0\}$

For $e \in \mathcal{E}_\Sigma$ we define the specified language $[\![e]\!] \subseteq \Sigma^*$ inductively by:

$$\begin{aligned}
[\![\epsilon]\!] &= \{\epsilon\} \\
[\![a]\!] &= \{a\} \\
[\![e^*]\!] &= ([\![e]\!])^* \\
[\![e_1 | e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot e_2]\!] &= [\![e_1]\!] \cdot [\![e_2]\!]
\end{aligned}$$

Finite Automata

Michael Rabin

Dana Scott

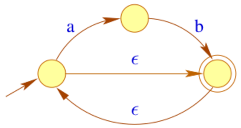
Definition Finite Automata

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

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

Finite Automata

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



Finite Automata

Once again, more formally:

- We define the transitive closure δ^* of δ as the smallest set δ' with:

  $(p, \epsilon, p) \in \delta'$ and
  $(p, xw, q) \in \delta'$ if $(p, x, p_1) \in \delta$ and $(p_1, w, q) \in \delta'$.

  δ^* 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:

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