Script generated by TTT

Title: Petter: Compilerbau (12.04.2018)

Date: Thu Apr 12 14:15:57 CEST 2018

Duration: 93:40 min

Pages: 15

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compiler Construction I

Dr. Michael Petter

SoSe 2018

Organizing

ccc.in.tum.de
www2.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

Dates:

Lecture: Thursday 14:15-15:45
Tutorial: Tbd in doodle until *Fr. 12th 12:00*

Exam:

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

Mini Seminars:

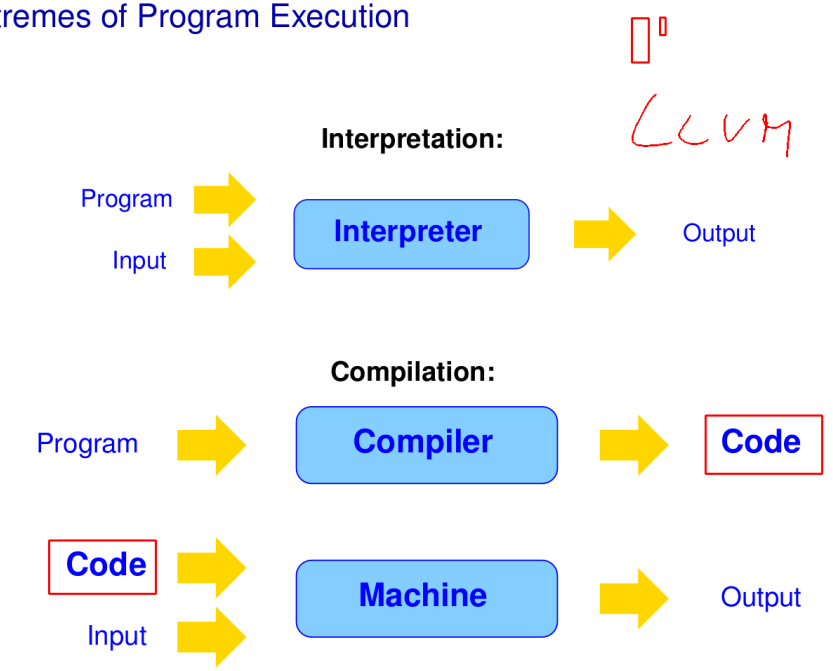
A short talk on a specific compiler related topic, e.g.

- Introduction to parser combinators
- Pager's LR(k):
  - Lane Tracing
  - Edge Pushing
- Type Inference
  - Hindley-Milner Type Systems
  - W Algorithm
- Single Static Assignment Form
- Register Allocation

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

Extremes of Program Execution



Interpretation vs. Compilation

Interpretation

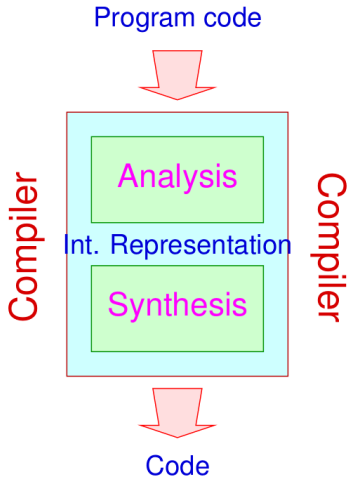
- No precomputation on program text necessary
  ⇒ no/small startup-overhead
- More context information allows for specific aggressive optimization

Compilation

- Program components are analyzed once, during preprocessing, instead of multiple times during execution
  ⇒ smaller runtime-overhead
- Runtime complexity of optimizations less important than in interpreter

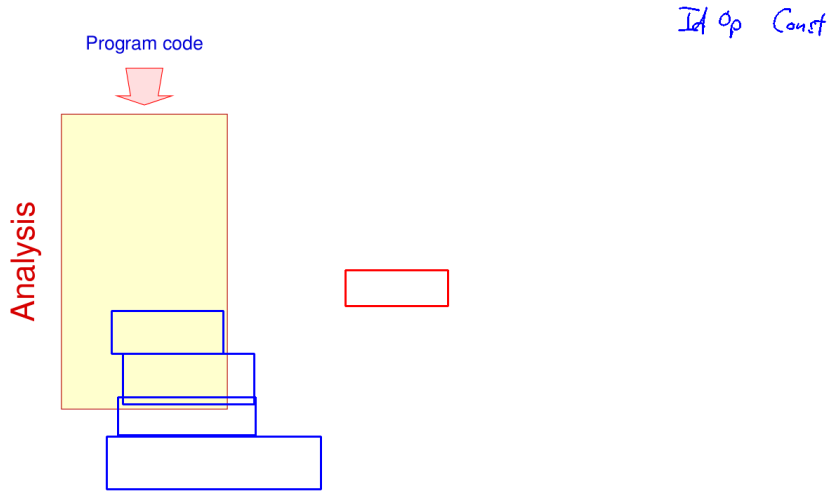
Compiler

general Compiler setup:



Compiler

The Analysis-Phase consists of several subcomponents:



The Lexical Analysis

Classified tokens allow for further pre-processing:

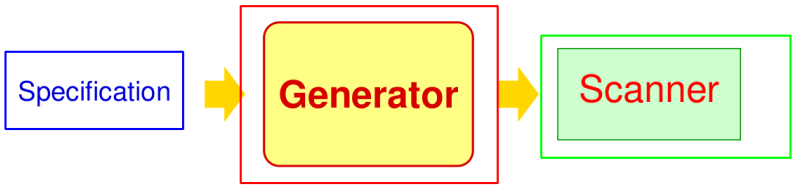
- Dropping irrelevant fragments e.g. Spacing, Comments,...
- Collecting Pragmas, i.e. directives for the compiler, often implementation dependent, directed at the code generation process, e.g. OpenMP-Statements;
- Replacing of Tokens of particular classes with their meaning / internal representation, 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:

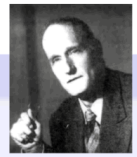
... in our case:

Chapter 1:
Basics: Regular Expressions

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 $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), e_1^* \in \mathcal{E}$ 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:

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