Script **generated by TTT**

Title: Petter: Compilerbau (11.04.2016)

Date: Mon Apr 11 14:23:19 CEST 2016

Duration: 92:37 min

Pages: 44

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compiler Construction I

Dr. Michael Petter

SoSe 2016

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

Aug 7th

Mini Projects:

- A practical implementation, based on a compiler fragment
- Implement a subcomponent:
  - Type system (memory model)
  - Typecasts
  - Type verification
  - Additional Language Features (ellipsis, 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 syntaxanalysis
- Attribute systems
- Typechecking
- Codegeneration for register machines
- Register assignment
- Optional: Basic optimization

Topic:

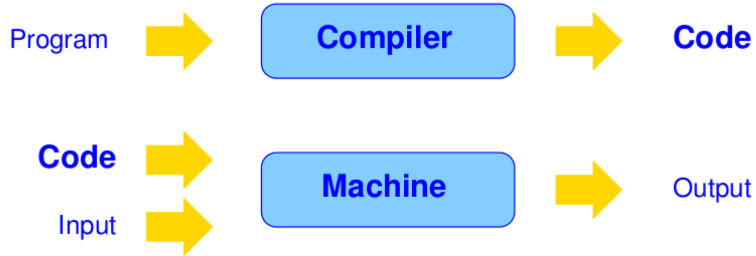
Introduction

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

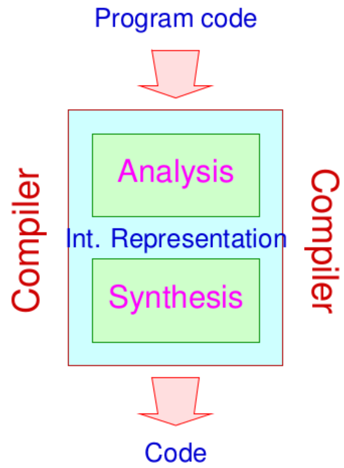
The Translation costs time

Advantage

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

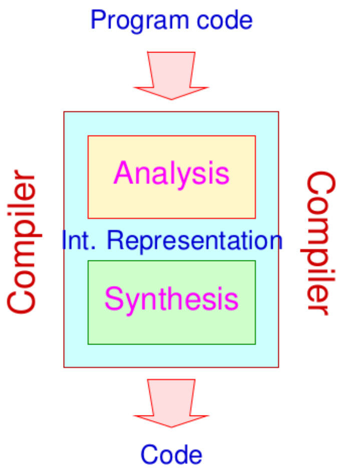
Compiler

general Compiler setup:



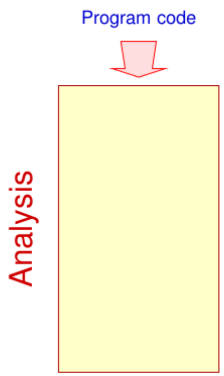
Compiler

general Compiler setup:



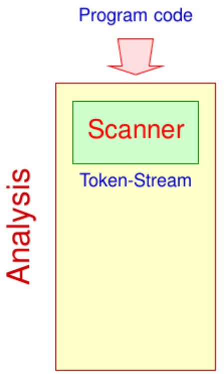
Compiler

The Analysis-Phase consists of several subcomponents:



Compiler

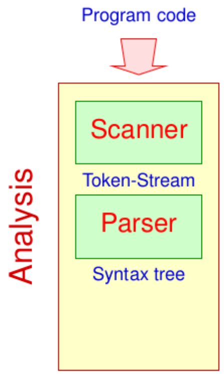
The Analysis-Phase consists of several subcomponents:



lexicographic Analysis:
Partitioning in tokens

Compiler

The Analysis-Phase consists of several subcomponents:

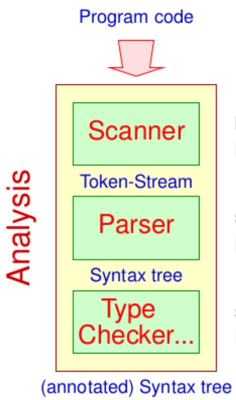


lexicographic Analysis:
Partitioning in tokens

syntactic Analysis:
Detecting hierarchical structure

Compiler

The Analysis-Phase consists of several subcomponents:



lexicographic Analysis:
Partitioning in tokens

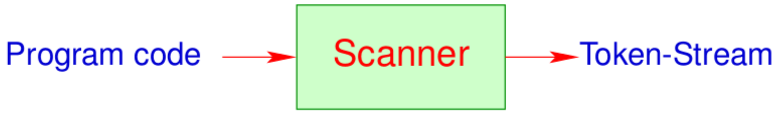
syntactic Analysis:
Detecting hierarchical structure

semantic Analysis:
Infering semantic properties

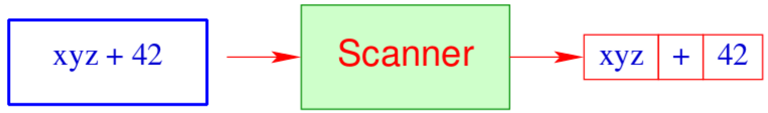
Topic:

Lexical Analysis

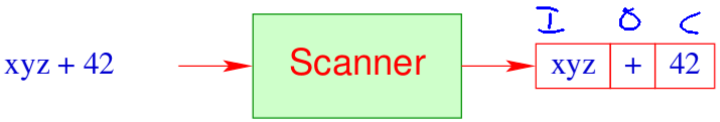
The Lexical Analysis



The Lexical Analysis



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. `xyz`, `pi`, ...
  - → 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:

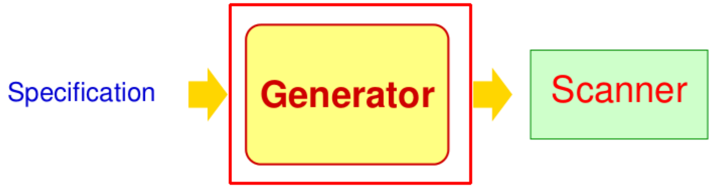
- Dropping irrelevant fragments e.g. Spacing, Comments,...
- Collecting Pragmas, i.e. directives for the compiler, which are not directly part of the source language, like OpenMP-Statements;
- Replacing of Tokens of particular classes with their meaning / internal representation, e.g.
  - → Constants;
  - → 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.

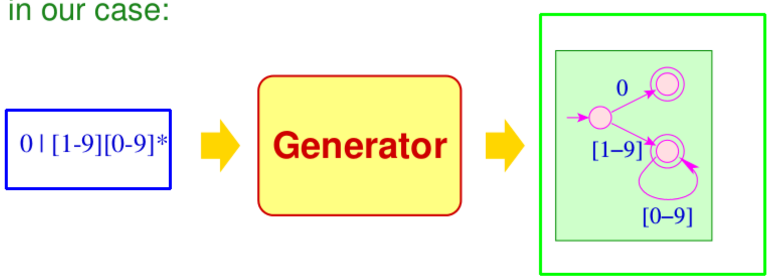


The Lexical Analysis - Generating:

... in our case:

The Lexical Analysis - Generating:

... in our case:



Specification of Token-classes: Regular expressions;
Generated Implementation: Finite automata + X

Regular Expressions

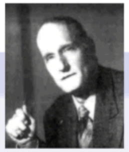
Basics

- 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.

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

... 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:

$$^* > \cdot > |$$

  and omit “$\cdot$”

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:

$$^* > \cdot > |$$

  and omit “$\cdot$”
- Real Specification-languages offer additional constructs:

$$e? \equiv (\epsilon \mid e)$$
$$e^+ \equiv (e \cdot e^*)$$

  and omit “ϵ”

Regular Expressions

Specification needs Semantics

...Example:

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

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

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

Keep in Mind:

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

$$\begin{array}{lll} (L)^* & = & \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 & = & \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{array}$$

Regular Expressions

Specification needs Semantics

...Example:

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

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

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

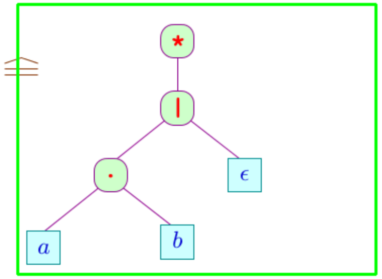
Keep in Mind:

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

$$\begin{array}{lll} (L)^* & = & \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 & = & \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{array}$$

- Regular expressions are internally represented as annotated ranked trees:

$(ab|\epsilon)^*$ $\Longleftarrow$



Inner nodes: Operator-applications;
Leaves: particular symbols or ϵ.

Regular Expressions

Example: Identifiers in Java:

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


Regular Expressions

Example: Identifiers in Java:

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

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


Regular Expressions

Example: Identifiers in Java:

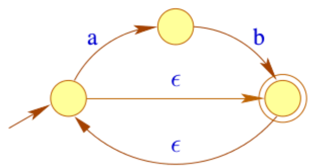
```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*

Float = {di}*(\.{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;
Lables: Consumed input;

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


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)

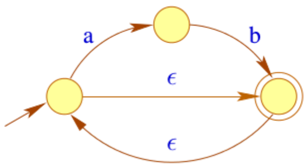
For an NFA, we reckon:

Definition Deterministic Finite Automata

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

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 $(p_1, w, q) \in \delta'$.

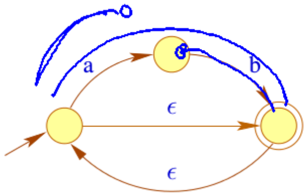
  δ^* characterizes for two states p and q the words, along each path between them

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

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

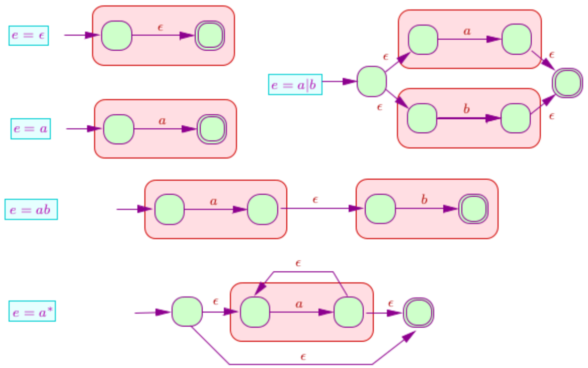
Finite Automata

- 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 ...



Chapter 3:
Converting Regular Expressions to NFAs

In Linear Time from Regular Expressions to NFAs




Ken Thompson

Thompson's Algorithm

Produces $\mathcal{O}(n)$ states for regular expressions of length n.

Berry-Sethi Approach


Gerard Berry


Ravi Sethi

Berry-Sethi Algorithm

Produces exactly $n + 1$ states without ϵ-transitions and demonstrates → *Equality Systems* and → *Attribute Grammars*

Idea:

The automaton tracks (conceptionally via a marker "•"), in the syntax tree of a regular expression, which subexpressions in e are reachable consuming the rest of input w.

Berry-Sethi Approach


Viktor M. Glushkov

Glushkov Algorithm

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

Idea:

The automaton tracks (conceptionally via a marker "•"), in the syntax tree of a regular expression, which subexpressions in e are reachable consuming the rest of input w.