Script generated by TTT

Title: Simon: Compilerbau (15.04.2013)

Date: Mon Apr 15 14:16:31 CEST 2013

Duration: 87:42 min

Pages: 40

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
  - Hauptseminars

Materials:

- TTT-based lecture recordings
- the slides
- Related literature list online
- Tools for visualization of virtual machines
- Tools for generating components of Compilers

Organizing

Zeiten:

Lecture: Mo. 14:15-15:45
Tutorial: Tuesday morning and afternoon

Exam

- Exam managed via TUM-online
- Successful tutorial exercises earns 0.3 bonus

Preliminary content

- Basics in regular expressions and automata
- Specification with regular expressions and implementation with automata
- Reduced context free grammars and pushdown automata
- Bottom-Up Syntaxanalysis
- Attribute systems
- Typechecking
- Codegeneration for stack machines
- Register assignment
- Basic Optimization

Themengebiet:

Introduction

Interpreter

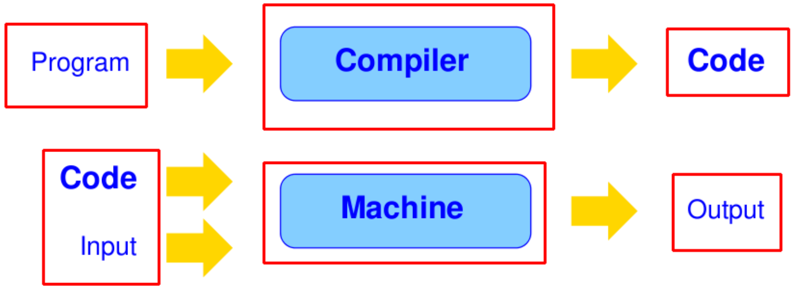
Pro:

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

Compiler

A precomputation on the program allows

- a more sophisticated variable management
- discovery and implementation of global optimizations

Disadvantage

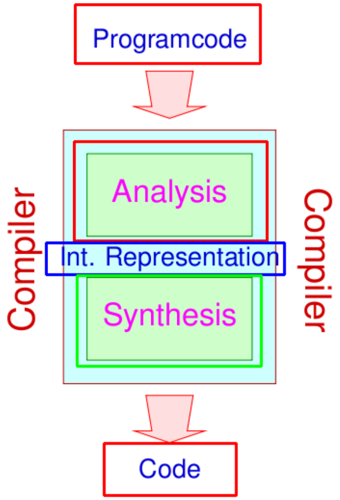
The Translation costs time

Advantage

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

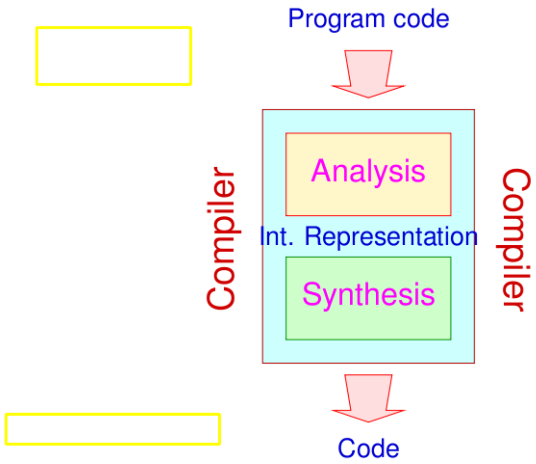
Compiler

general Compiler setup:



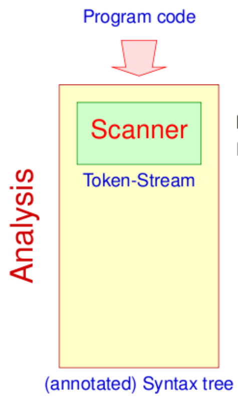
Compiler

general Compiler setup:



Compiler

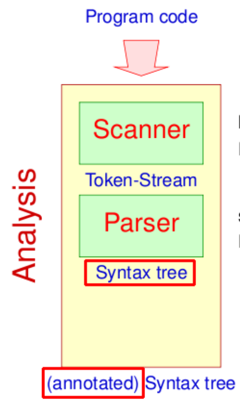
The Analysis-Phase is divided in several parts:



lexicographic Analysis:
Partitioning in tokens

Compiler

The Analysis-Phase is divided in several parts:

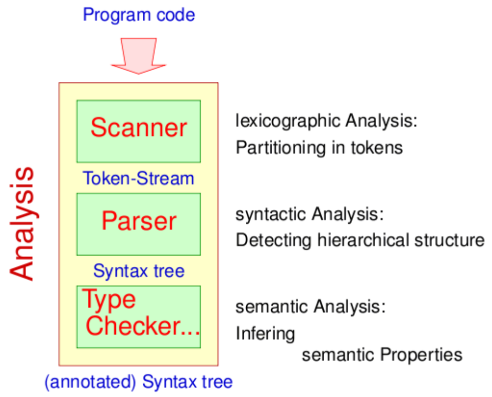


lexicographic Analysis:
Partitioning in tokens

syntactic Analysis:
Detecting hierarchical structure

Compiler

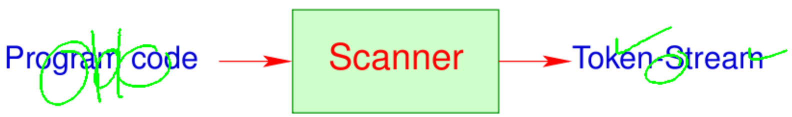
The Analysis-Phase is divided in several parts:



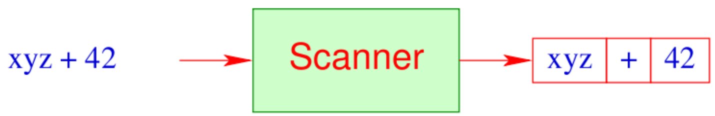
Themengebiet:

Lexical Analysis

The lexical Analysis

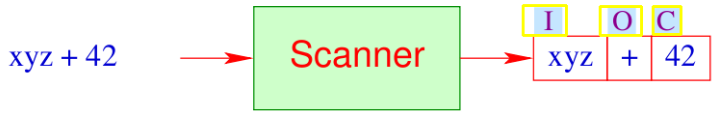


The lexical Analysis



- 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



- 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,...
- 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, evt. compared to reserved terms (if not already done by the scanner) and possibly replaced with an index.

⇒ 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:

Advantages

Productivity	The component can be produced more rapidly
Correctness	The component implements (provably) the specification.
Efficiency	The generator can provide the produced component with very efficient algorithms.

The lexical Analysis - Generating:

Advantages

Productivity	The component can be produced more rapidly
Correctness	The component implements (provably) the specification.
Efficiency	The generator can provide the produced component with very efficient algorithms.

Disadvantages

- Specification is just another form of programming — admittedly possibly simpler
- Generation instead of implementatation pays off for Routine-tasks only
  ... and is only good for problems, that are well understood

The lexical Analysis - Generating:

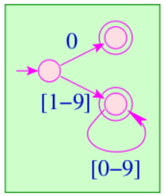
... in our case:

Specification → **Generator** → Scanner

The lexical Analysis - Generating:

... in our case:

0 | [1-9][0-9]* → **Generator** →



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

Kapitel 1:

Basics: Regular Expressions

Regular expressions

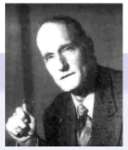
Basics

- 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.

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 $a, 0, \$, \ldots$ and Meta-symbols $(, |,), \ldots$
- To avoid (ugly) parantheses, we make use of Operator-Precedences:

$$^* > \cdot > |$$

and omit "$\cdot$"
- Real Specifications-languages offer additional constructs:

$$\begin{array}{lcl} e? & \equiv & (\epsilon \mid e) \\ e^+ & \equiv & (e \cdot e^*) \end{array}$$

and omit "ϵ"

Regular expressions

Specifications need 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:

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

Beachte:

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

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

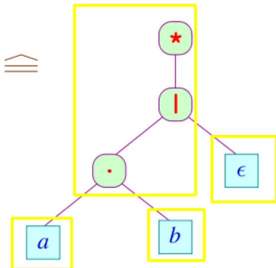
Beachte:

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

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

- Regular expressions are internally represented as annotated ranked trees:

$(ab|\epsilon)^*$ $\widehat{=}$



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:

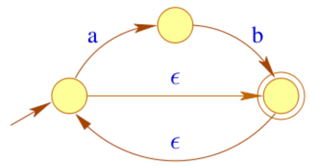
```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*

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

A no~~nich~~t-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

A nonicht-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:

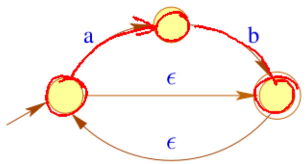
Definition

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

Finite automata

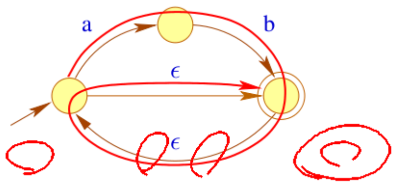
- Computations are paths in the graph.
- Accepting computations lead from I to F.
- An accepted word is the sequence of lables along an accepting computation ...

ab



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 ...



Kapitel 3:

Converting Regular expressions in NFAs