TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

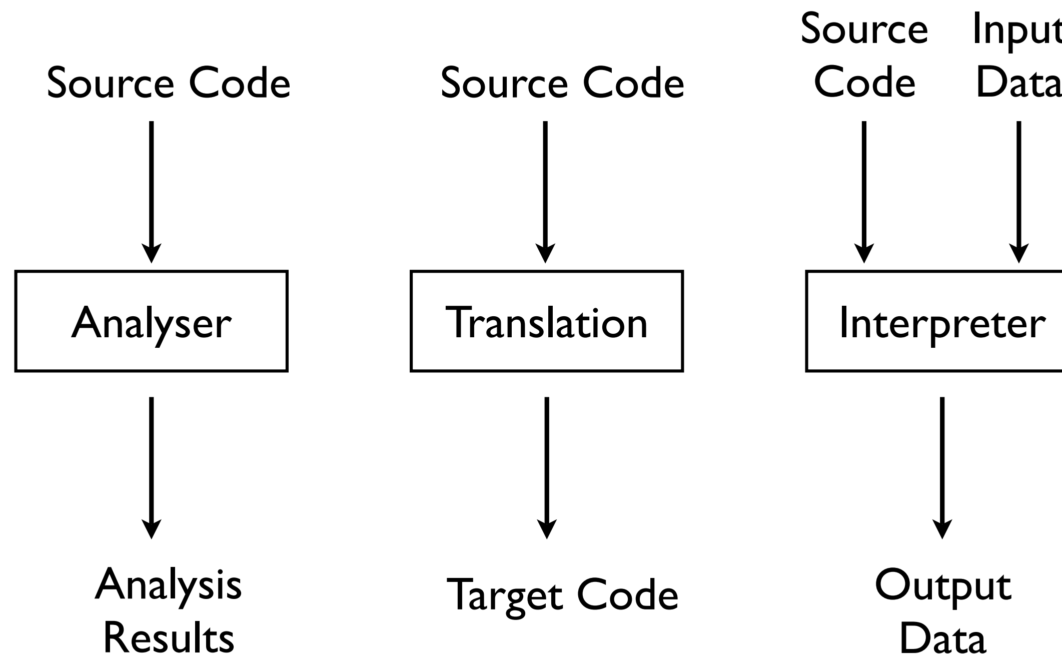
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Outline

1. Übersetzer und Werkzeuge zur Sprachverarbeitung
   - Examples
2. Sprachverarbeitung
   - Terminology and Requirements
   - Compiler Architecture
3. Scannen und Parsen
   - Scanner
   - Parser

Übersetzer und Werkzeuge zur Sprachverarbeitung

Tasks of Language-Processing Tools I



Analysis, translation and interpretation are often combined.

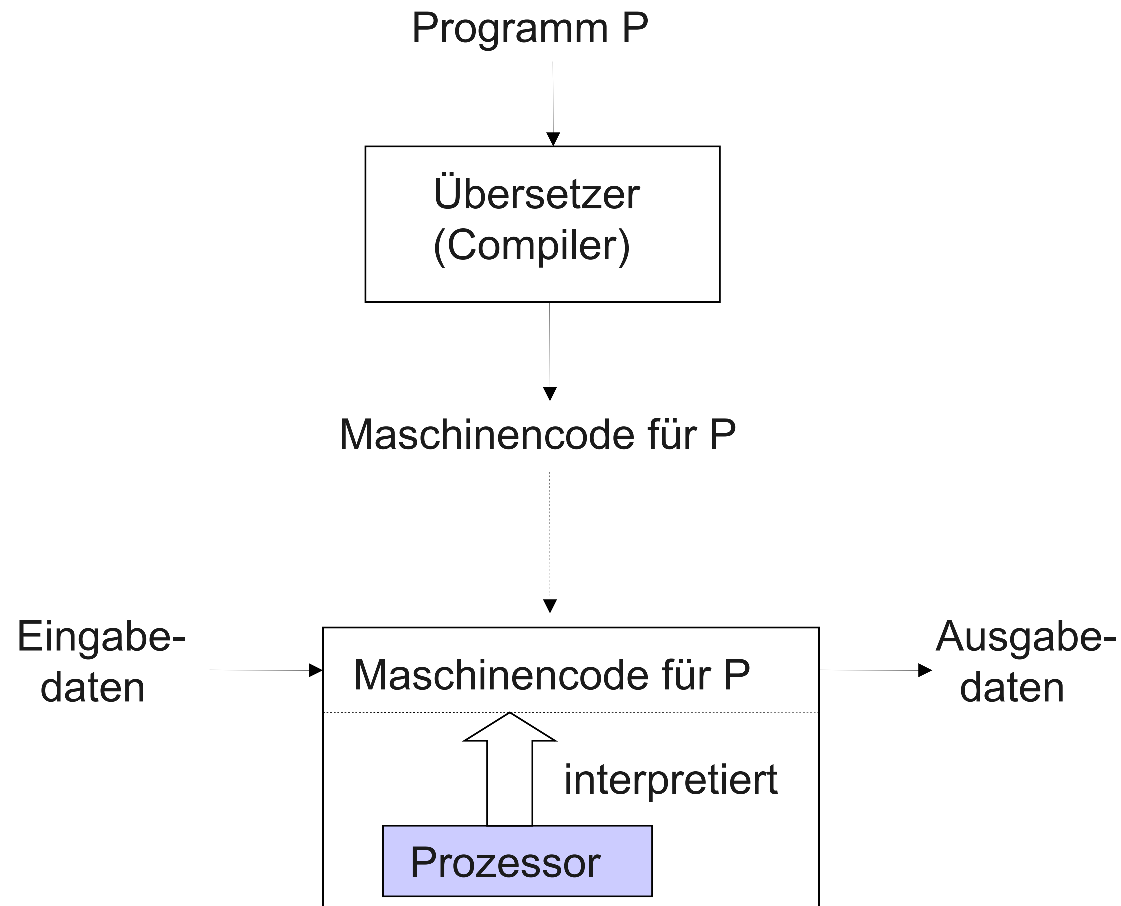
Tasks of Language-Processing Tools II

1. **Translation**
   - Compiler implements analysis and translation
   - OS and real machine implement interpretation

   Pros:

   - Most efficient solution
   - One interpreter for different programming languages
   - Prerequisite for other solutions

Tasks of Language-Processing Tools III



Tasks of Language-Processing Tools IV

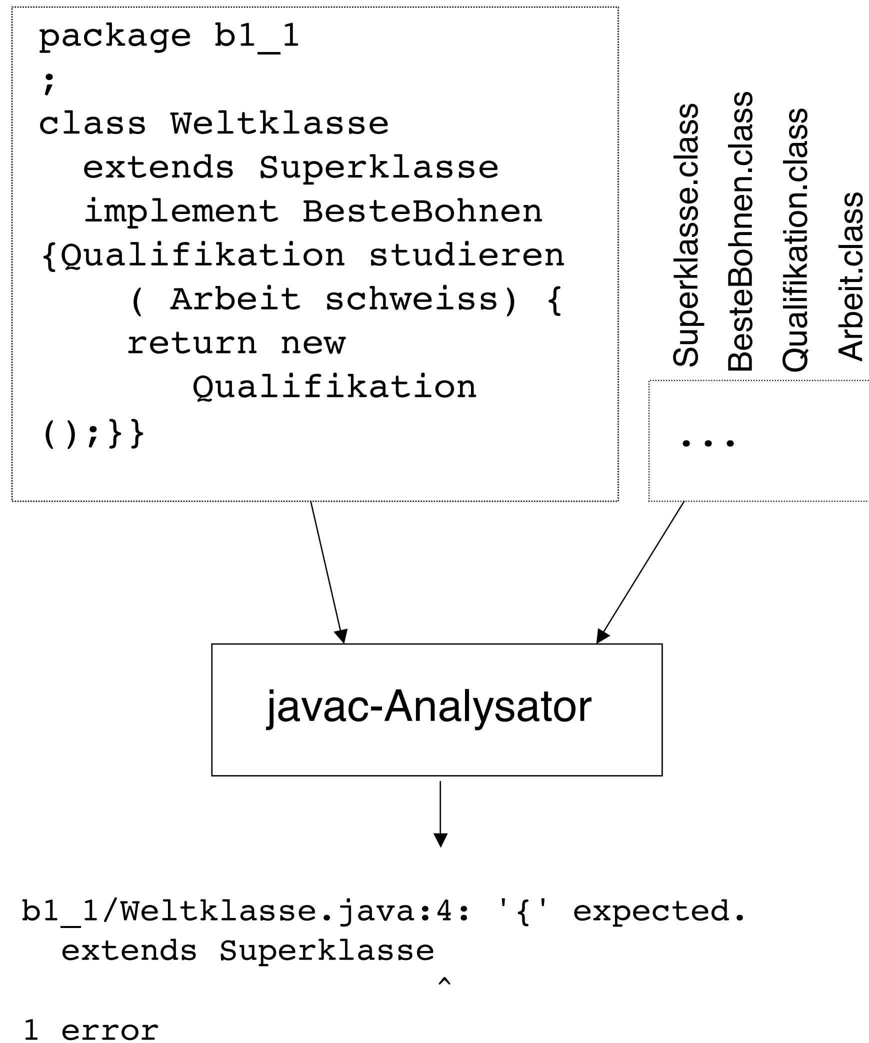
2. **Direct interpretation**
   - Interpreter implements all tasks.
   - Examples: JavaScript, command line languages (bash)
   - Pros: No translation necessary (but analysis at run-time)

Tasks of Language-Processing Tools V

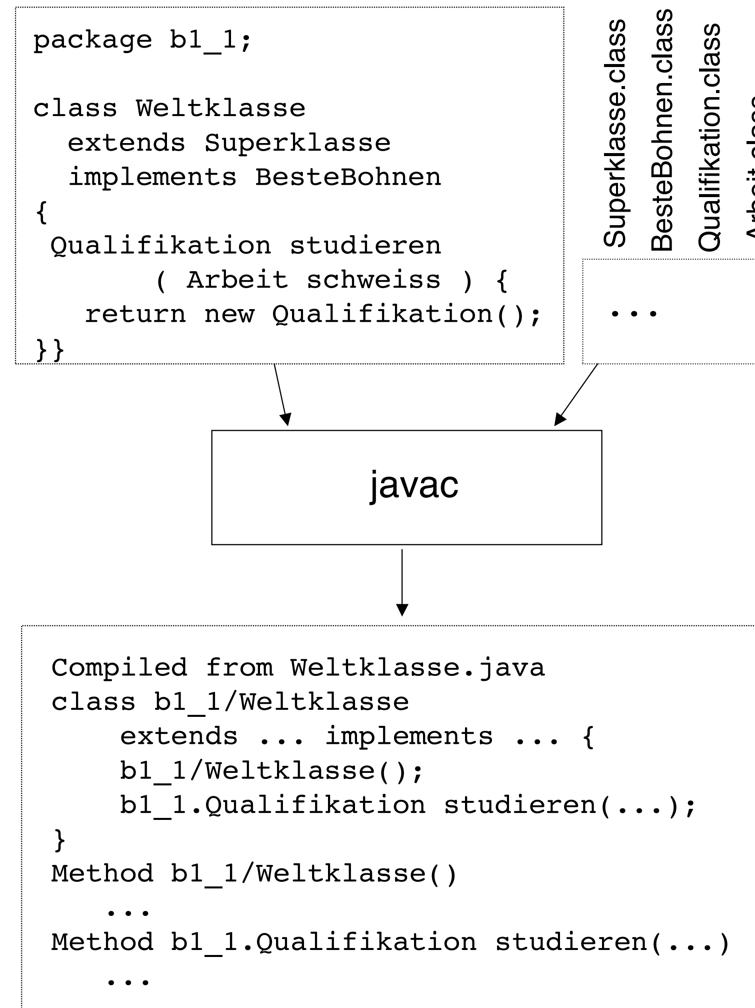
3. **Abstract and virtual machines**
   - Compiler implements analysis and translation to abstract machine code
   - Abstract machine works as interpreter
   - Examples: Java/JVM, C#, .NET
   - Pros:
     - Platform independent (portability, mobile code)
     - Self-modifing programs possible
4. **Other combinations**

Examples

Example: Analysis



Example 1: Translation I



Example 1: Translation II

Result of translation

```
Compiled from Weltklasse.java
class b1_1/Weltklasse
      extends b1_1.Superklasse
      implements b1_1.BesteBohnen {
    b1_1/Weltklasse();
    b1_1.Qualifikation studieren(b1_1.Arbeit);
}

Method b1_1/Weltklasse()
   0 aload_0
   1 invokespecial #6 <Method b1_1.Superklasse()>
   4 return

Method b1_1.Qualifikation studieren(b1_1.Arbeit)
   0 new #2 <Class b1_1.Qualifikation>
   3 dup
   4 invokespecial #5 <Method b1_1.Qualifikation()>
   7 areturn
```

Example 2: Translation I

```
int main() {
  printf("Willkommen zur Vorlesung!");
  return 0;
}
```

↓

gcc

↓

```
  .file       "hello_world.c"
  .version    "01.01"
gcc2_compiled.:
.section      .rodata
.LC0:
  .string     "Willkommen zur Vorlesung!"
.text
  .align 16
.globl main
  .type        main,@function
main:
  pushl %ebp
  movl %esp,%ebp
  subl $8,%esp

...
```

Example 2: Translation II

Result of translation

```
  .file       "hello_world.c"
  .version    "01.01"
gcc2_compiled.:
.section      .rodata
.LC0:
  .string     "Willkommen zur Vorlesung!"
.text
  .align 16
.globl main
  .type        main,@function
main:
  pushl %ebp
  movl %esp,%ebp
  subl $8,%esp
  addl $-12,%esp
  pushl $.LC0
  call printf
  addl $16,%esp
  xorl %eax,%eax
  jmp .L2
  .p2align 4,,7
.L2:
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size  main,.Lfe1-main
  .ident "GCC: (GNU) 2.95.2 19991024 (release)"
```

Example 3: Translation

groovy.tex (104 bytes)

```
\documentclass{article}
\begin{document}
\vspace*{7cm}
\centerline{\Huge\bf It's groovy}
\end{document}
```

latex

groovy.dvi (207 bytes, binary)

```
...
```

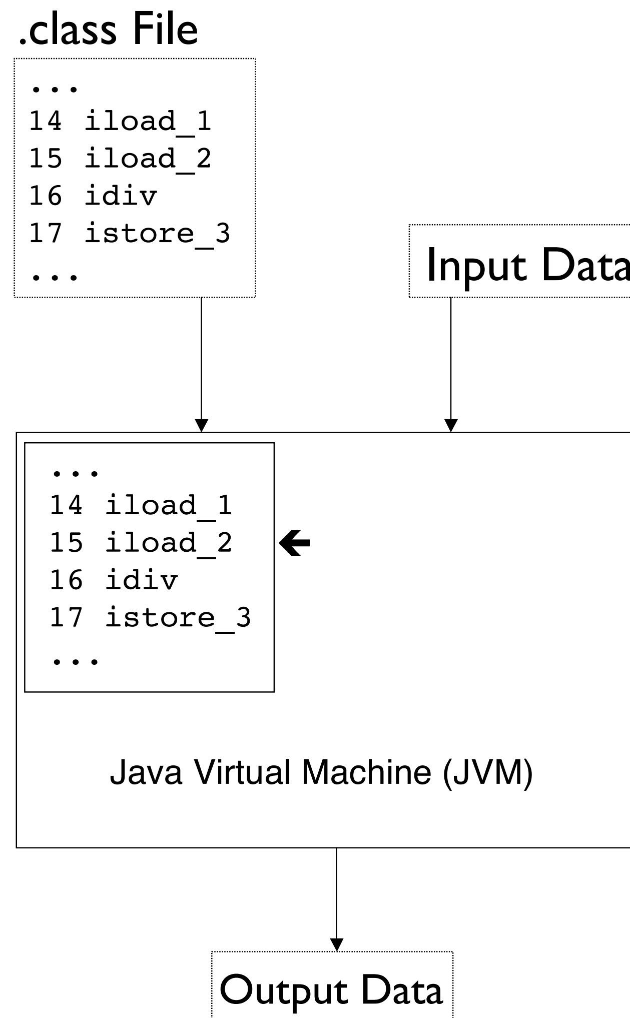
dvips

groovy.ps (7136 bytes)

```
%!PS-Adobe-2.0
%%Creator: dvips(k) 5.86 ...
%%Title: groovy.dvi
...
```

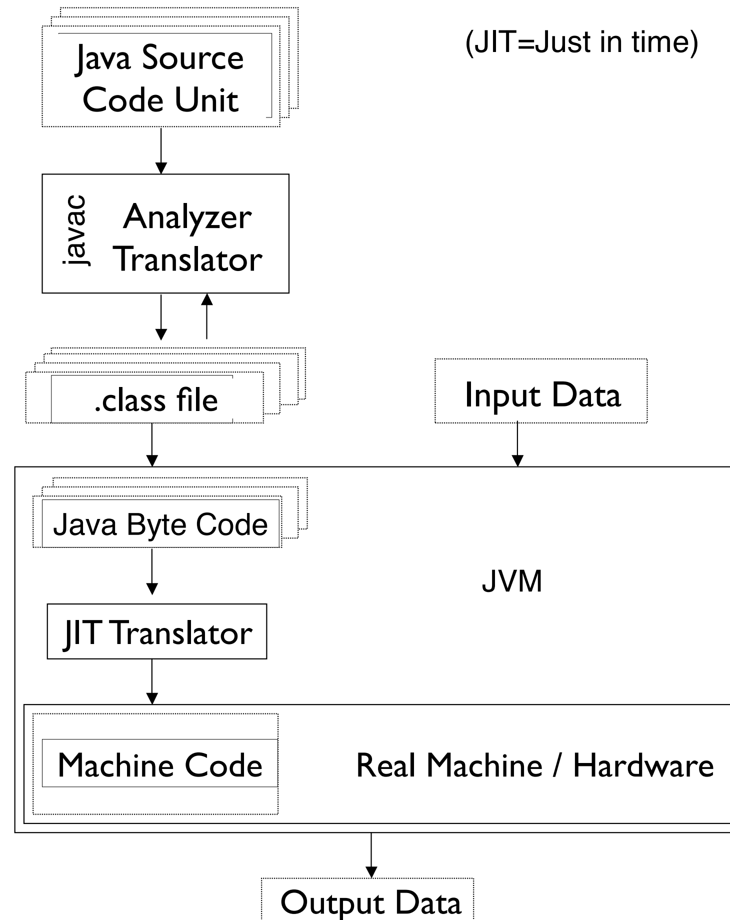
TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Example: Interpretation



Example: Combined technique

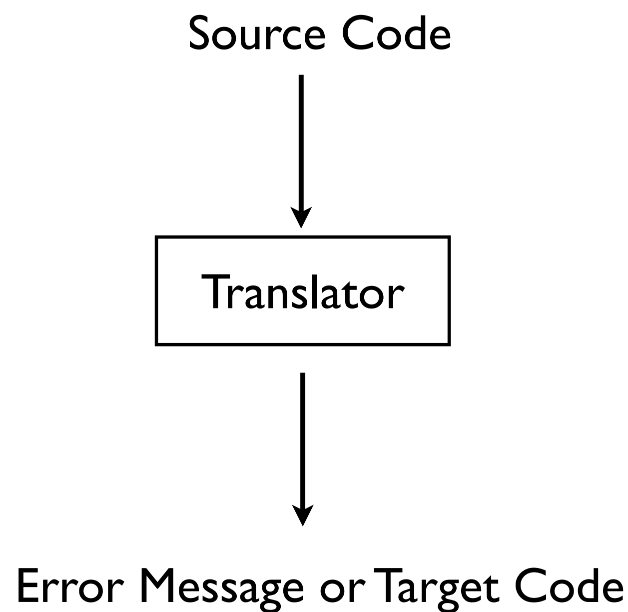
Java implementation with just-in-time (JIT) compiler



Sprachverarbeitung

Terminology and Requirements

Language processing: The task of translation



- **Translator** (in a broader sense):
  Analysis, optimization and translation
- **Source code**:
  Input (string) for translator in syntax of source language (SL)
- **Target Code**:
  Output (string) of translator in syntax of target language (TL)

Aspects of language processing

- Different inputs:
  - Program text
  - Specification
  - Diagrams
- Different goals:
  - Transformation (XSLT, refactoring)
  - Pretty printing, formatting
  - Semantic analysis (program comprehension)
  - Optimization
  - Translation/generation from one language into another

Compile time vs. run-time

- **Compile time**: during run-time of compiler/translator
  Static: All information/aspects known at compile time, e.g.:
  - Type checks
  - Evaluation of constant expressions
  - Relative addresses
- **Run-time**: during run-time of compiled program
  Dynamic: All information that are not statically known, e.g.:
  - Allocation of dynamic arrays
  - Bounds check of arrays
  - Dynamic binding of methods
  - Memory management of recursive procedures

For *dynamic aspects* that cannot be handled at *compile time*, the compiler generates code that handles these aspects at *run-time*.

What is a good compiler?

Requirements for translators

- Error handling (static/dynamic)
- Efficient target code
- Choice: Fast translation with slow code
  vs. slow translation with fast code
- Semantically correct translation

Semantically correct translation

Intuitive definition: Compiled program behaves according to language definition of source language.

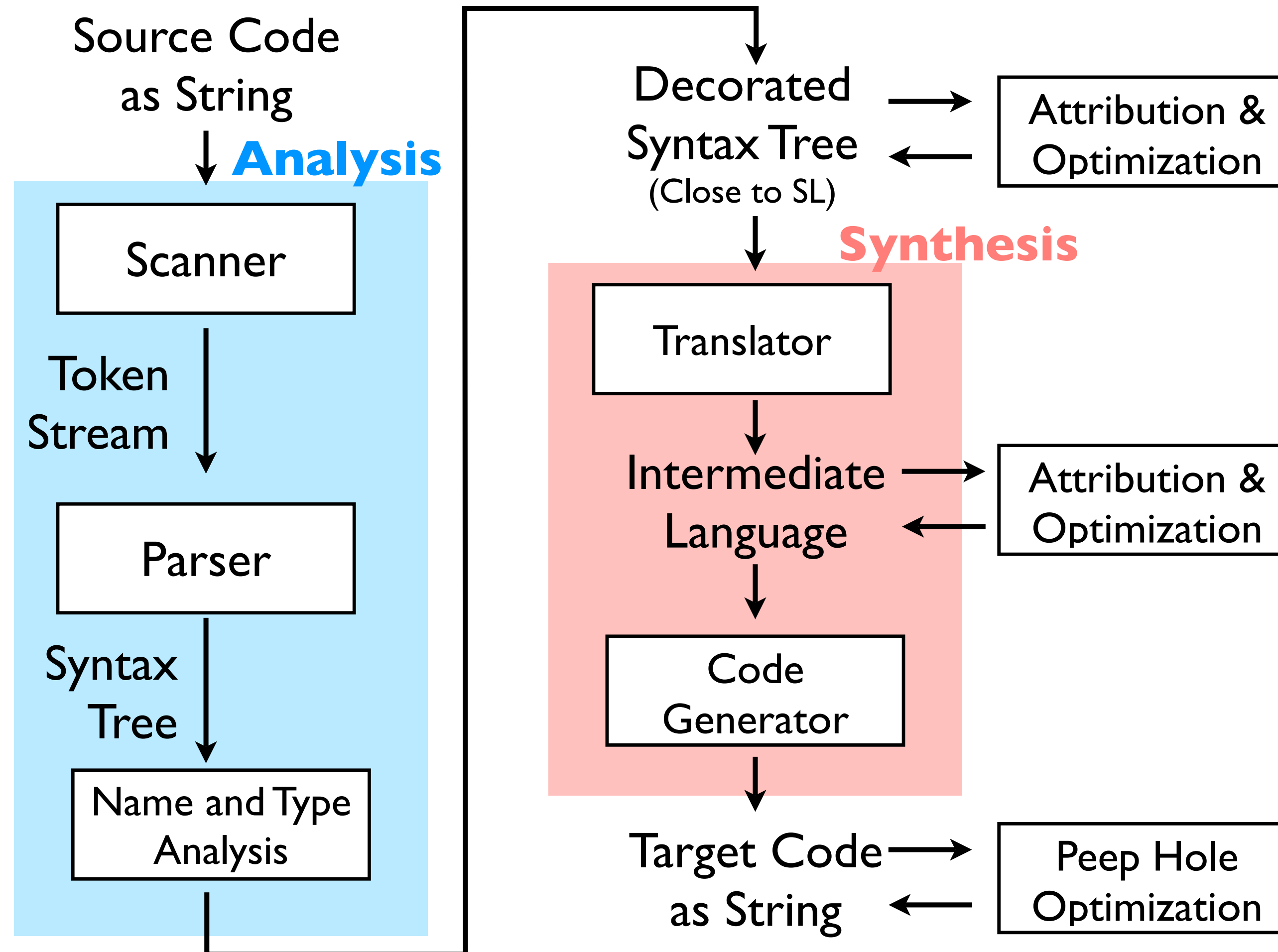
Formal definition:

- semSL: SL_Program $\times$ SL_Data $\rightarrow$ SL_Data
- semTL: TL_Program $\times$ TL_Data $\rightarrow$ TL_Data
- compile: SL_Program $\rightarrow$ TL_Program
- code: SL_Data $\rightarrow$ TL_Data
- decode: TL_Data $\rightarrow$ SL_Data

Semantic correctness:
semSL(P,D) = decode(semTL(compile(P), code(D)))

Compiler Architecture

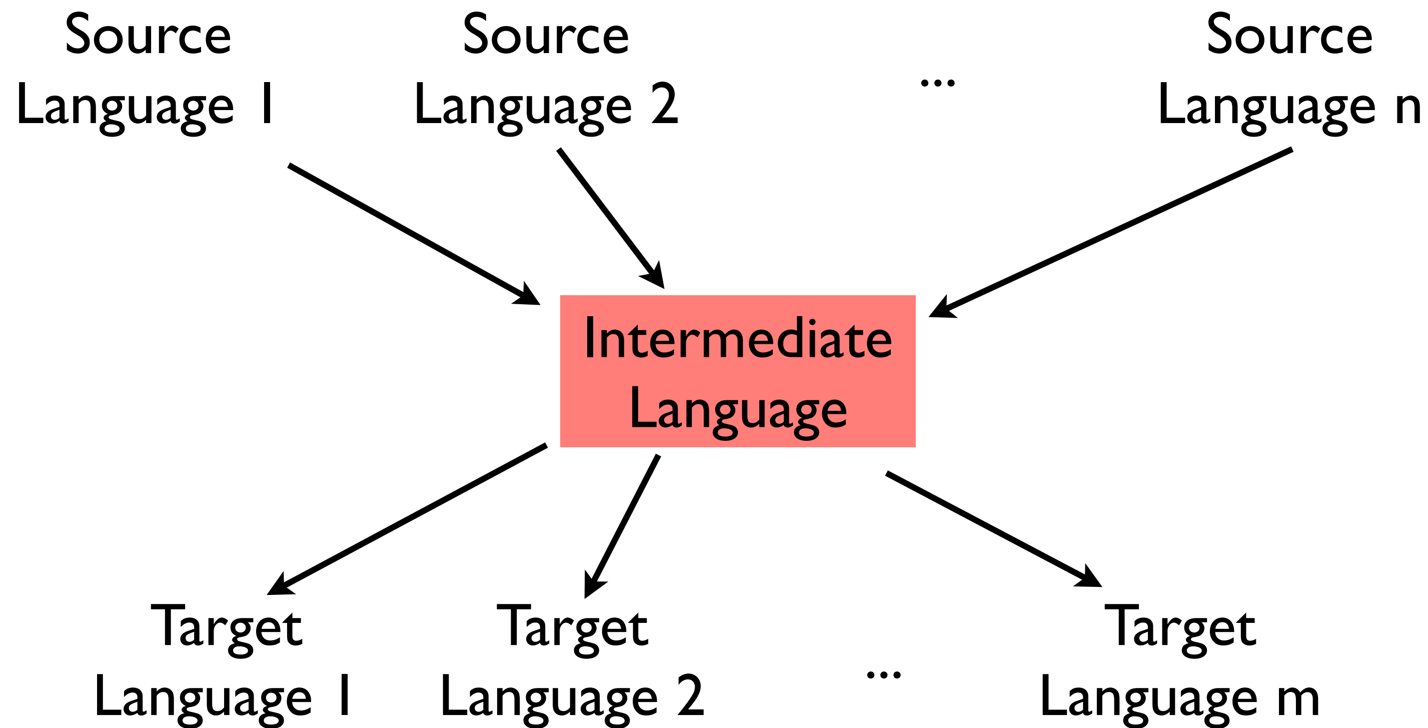
Compiler Architecture



Properties of compiler architectures

- Phases are conceptual units of translation
- Phases can be interleaved
- Design of phases depends on source language, target language and design decisions
- Phase vs. **pass** (phase can comprise more than one pass.)
- Separate translation of pogram parts
  (Interface information must be accessible.)
- Combination with other architecture decisions:
  Common intermediate language

Common intermediate language



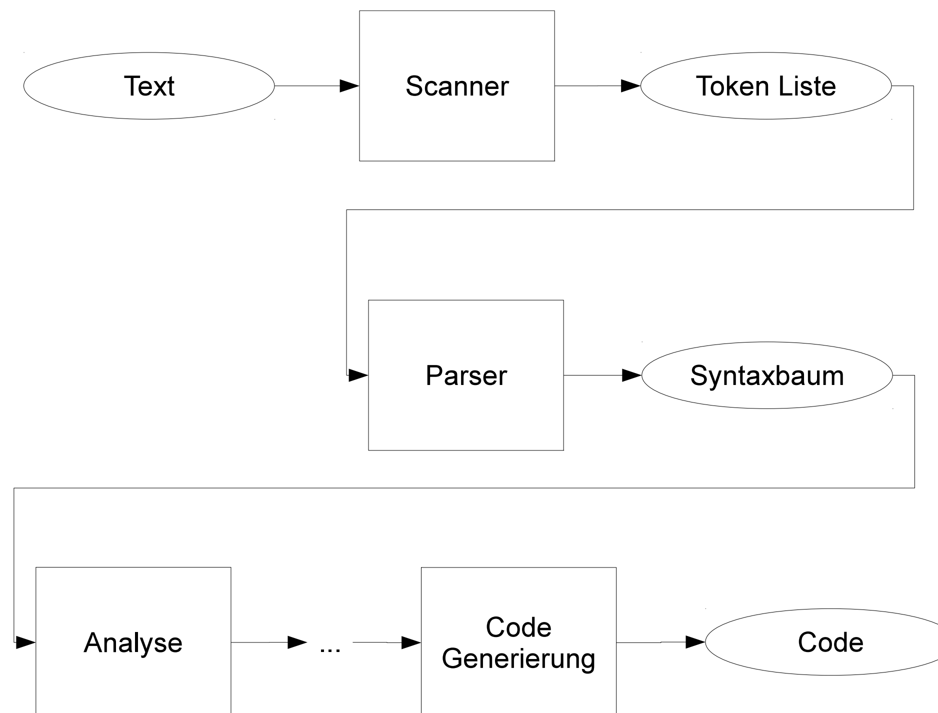
Dimensions of compiler construction

- Programming languages
  - Sequential procedural, imperative, OO-languages
  - Functional, logical languages
  - Parallel languages/language constructs
- Target languages/machines
  - Code for abstract machines
  - Assembler
  - Machine languages (CISC, RISC, ...)
  - Multi-processor/multi-core architectures
  - Memory hierarchy
- Translation tasks: analysis, optimization, synthesis
- Construction techniques and tools: bootstrapping, generators
- Portability, specification, correctness

Scannen und Parsen

Scannen und Parsen I

Die ersten Phasen des Übersetzers analysieren die Zeichenreihe und erzeugen eine Strukturbaum gemäß der Grammatik.



Scannen und Parsen II

Für einfache Beispiele ist die Implementierung von Scanner und Parser relativ einfach

Für komplexe Grammatiken ist die manuelle Implementierung schwierig und fehleranfällig; man möchte Unterstützung in Form von Generatoren haben.

Scanner-Generatoren nehmen Reguläre Ausdrücke als Eingabe und erzeugen Programmcode für einen Scanner.

Parser-Generatoren nehmen kontextfreie Grammatiken als Eingabe und erzeugen Programmcode für einen Parser.

Scanner- und Parser-Generatoren gibt es für viele Programmiersprachen, unter anderem auch für Java.

Scanner

JFlex I

Ein Scanner wandelt einen Text in eine Liste von Tokens um.

JFlex ist ein Scanner-Generator für Java.

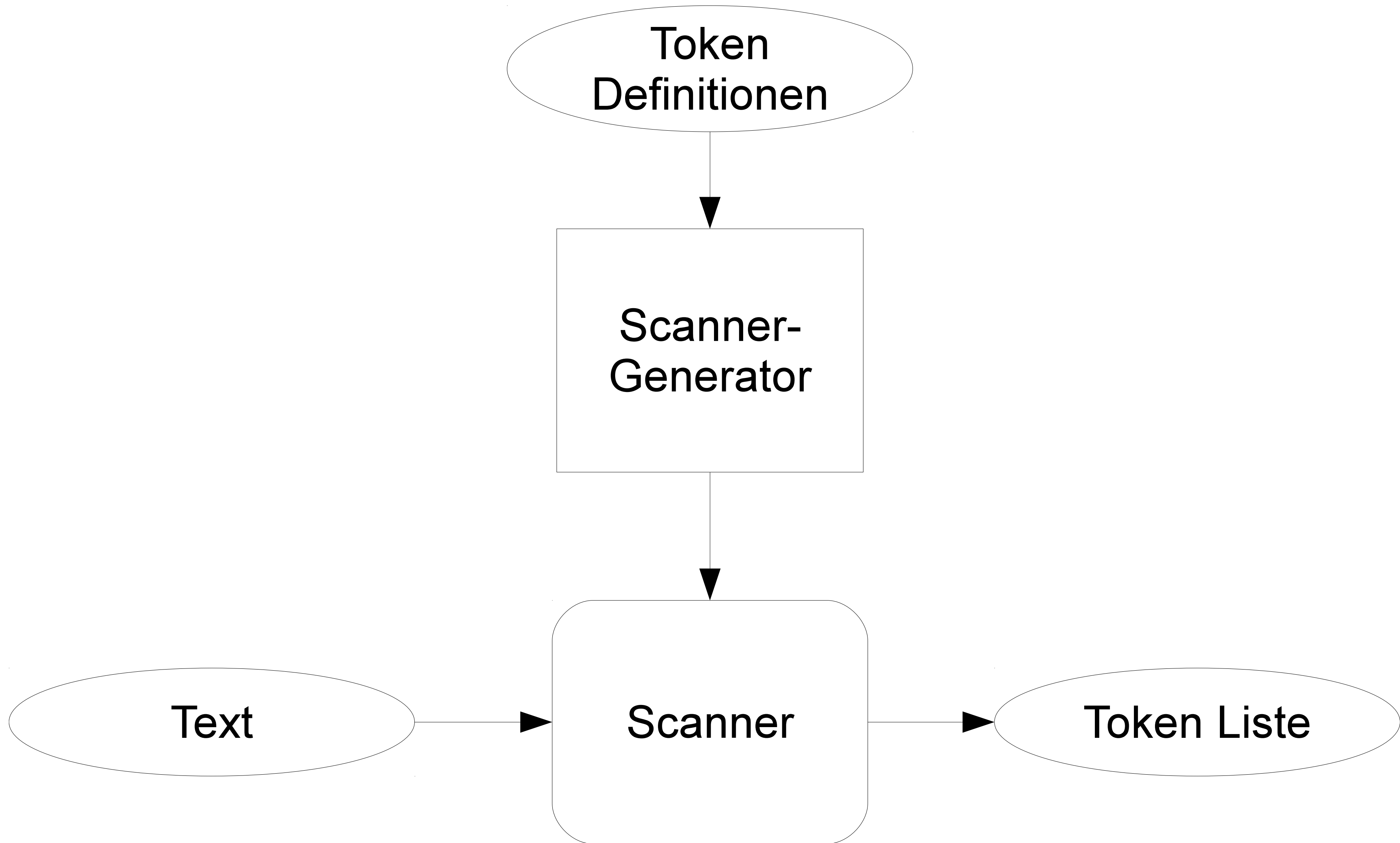
Entwickelt wurde er von Elliot Berk an der Princeton University, New Jersey, USA.

Die Implementierung ist in Java geschrieben und generiert Scanner für Java Programme.

Die Anbindung an einen Parser erfolgt mittels Java Typen und Objekte.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

JFlex II



JFlex Beispiel I

```
%cup
%line
%column
%unicode
%class Scanner

letter              = [A-Za-z]
digit               = [0-9]
alphanumeric        = {letter}|{digit}
other_id_char       = [_]
identifier          = {letter}({alphanumeric}|{other_id_char})*
integer             = {digit}*
whitespace          = [ \n\t\r]
```

JFlex Beispiel II

```
%%
"*"          {return new Symbol(sym.TIMES, yyline, yycolumn);}
"+"          {return new Symbol(sym.PLUS, yyline, yycolumn); }
"("          {return new Symbol(sym.LPAR, yyline, yycolumn); }
")"          {return new Symbol(sym.RPAR, yyline, yycolumn); }
{identifier} {return new Symbol(sym.IDENT, yyline, yycolumn,
                                yytext()); }
{integer}    {return new Symbol(sym.INT, yyline, yycolumn,
                                yytext()); }
{whitespace} {/* Ignore whitespace. */ }
```

Parser

Parser-Generatoren I

Ein Parser wandelt eine Liste von Tokens gemäß einer Grammatik in einen Syntaxbaum um.

Die Grammatik muss dem Parser-Generator mitgeteilt werden. Bei vielen dieser Generatoren geschieht dies in Form einer Beschreibung in Backus-Naur-Form.

- Nichtterminale werden mit `::=` definiert
- Das Zeichen `|` dient als Alternative-Trennzeichen
- Aufeinander folgende Terminale und Nichtterminale werden durch Leerzeichen getrennt.

CUP Parser-Generator I

CUP ist ein Parser-Generator für Java.

Entwickelt wurde er von Scott E. Hudson, Frank Flannery, C. Scott Ananian, Don Wong und Andrew W. Appel (Princeton).

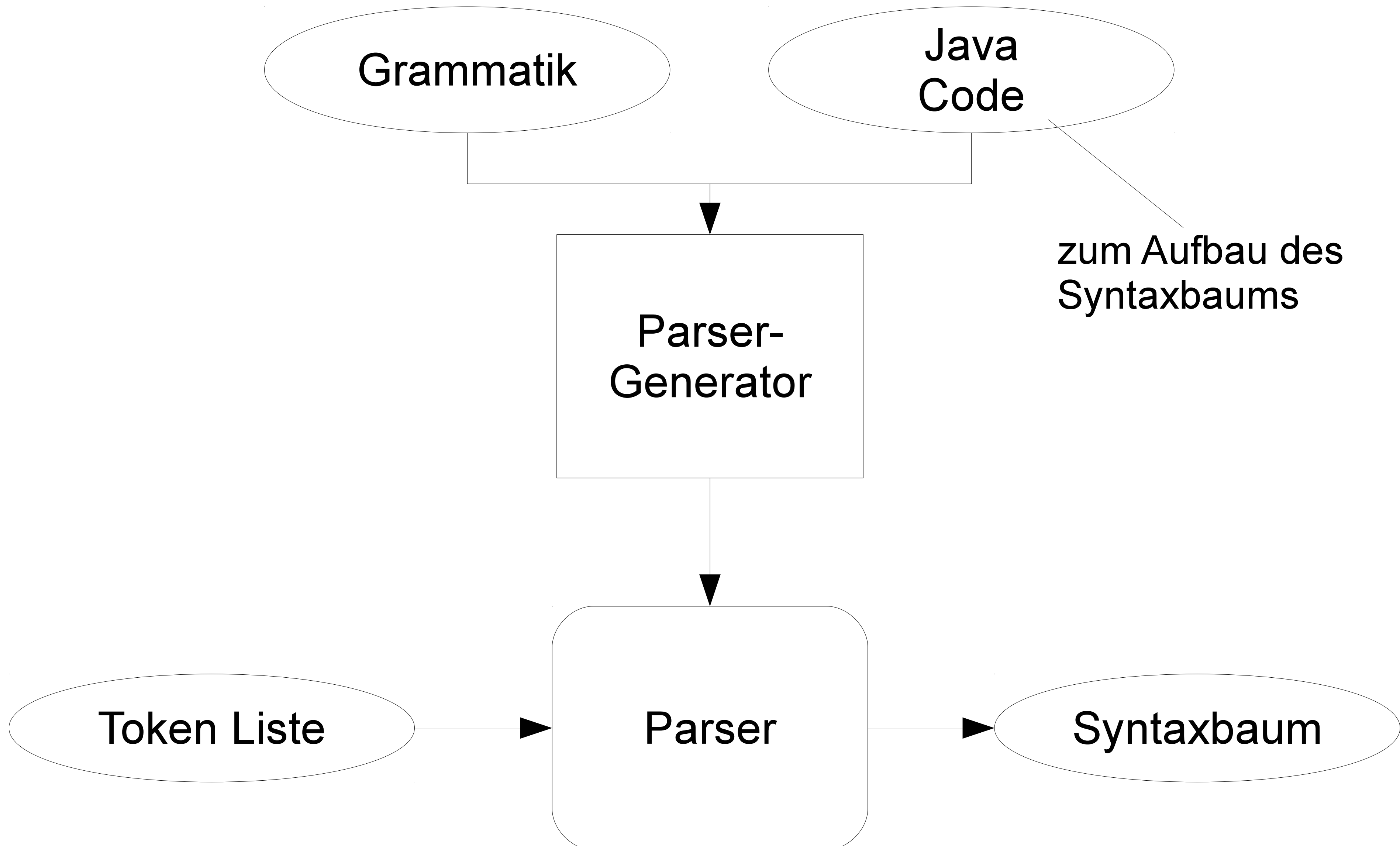
CUP benötigt einen externen Scanner, der in Java geschrieben ist, und erzeugt Parser in Java.

Eine Parser-Definitionsdatei besteht aus:

- Definition der Terminal- und Nichtterminal-Symbole (evtl. mit Java-Typen annotiert)
- Grammatik in (leicht modifizierter) Backus-Naur-Form
- Java-Code Blöcken für jede Produktion (zur Erzeugung des Syntaxbaums oder ähnlichem)

Die Verbindung zwischen Scanner und Parser erfolgt mit Hilfe von Objekten der Klasse `Symbol`.

CUP Parser-Generator II



CUP Beispiel I

```
terminal TIMES;
terminal PLUS;
terminal LPAR;
terminal RPAR;
terminal String INT;
terminal String IDENT;

non terminal Ausdruck exp;

precedence left PLUS;
precedence left TIMES;
```

CUP Beispiel II

```
start with exp;

exp ::= LPAR exp:e RPAR
        {: RESULT = e; :}
      | exp:e1 PLUS exp:e2
      {: RESULT = new Add(e1, e2); :}
      | exp:e1 TIMES exp:e2
      {: RESULT = new Mult(e1, e2); :}
      | INT:i
      {: RESULT = new Zahl(Integer.parseInt(i)); :}
      | IDENT:i
      {: RESULT = new Bzn(i); :}
      ;
```

Bemerkungen

Wir haben uns nur mit dem Compiler-Frontend (Scanner und Parser) beschäftigt.

Ein Übersetzer muss auf Basis des Syntaxbaums weitere Aufgaben erfüllen.

- Namensanalyse (z.B. Sichtbarkeit von Bindungen)
- Typanalyse (z.B. Typkorrektheit des Programms prüfen)
- Optimierung (z.B. Ersetzung von statisch auswertbaren Ausdrücken durch Konstanten)

Weitere Beispiele für Scanner-/Parser-Generatoren

Für Java:

- ANTLR (Scanner- und Parser-Generator)
- SableCC (Scanner- und Parser-Generator; erzeugt automatisch Syntaxbäume)

Für andere Sprachen:

- Alex (Scanner-Generator; Haskell)
- Happy (Parser-Generator; Haskell)
- Parsec (Parser combinator; Haskell)
- Flex (Scanner-Generator; C, C++)
- Bison (Parser-Generator; C, C++, Java)