Compilerbau

Martin Plümicke

WS 2021/22

Agenda

Überblick Vorlesung

- Literatur

Compiler Überblick

Compiler (I)

- Scanner
- Parser

Überblick Funktionale Programmierung

- Einleitung
- Haskell-Grundlagen
- Scanner in Funktionalen Sprachen
- Parser in Funktionalen Sprachen

Compiler (II)

- Abstrakte Syntax
- Abstrakte Syntax Java

Literatur


Bauer and Höllerer.
Übersetzung objektorientierter Programmiersprachen.
Springer-Verlag, 1998, (in german).


Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
Compiler: Prinzipien, Techniken und Werkzeuge.
Pearson Studium Informatik. Pearson Education Deutschland, 2. edition, 2008.
(in german).


Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques and Tools.
Addison Wesley, 1986.


Reinhard Wilhelm and Dieter Maurer.
Übersetzerbau.
Springer-Verlag, 2. edition, 1992.
(in german).

Literatur II

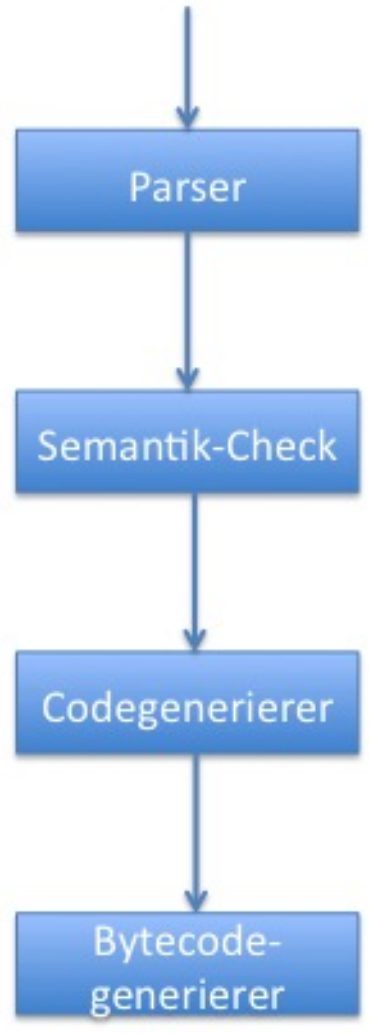

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java® Virtual Machine Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.

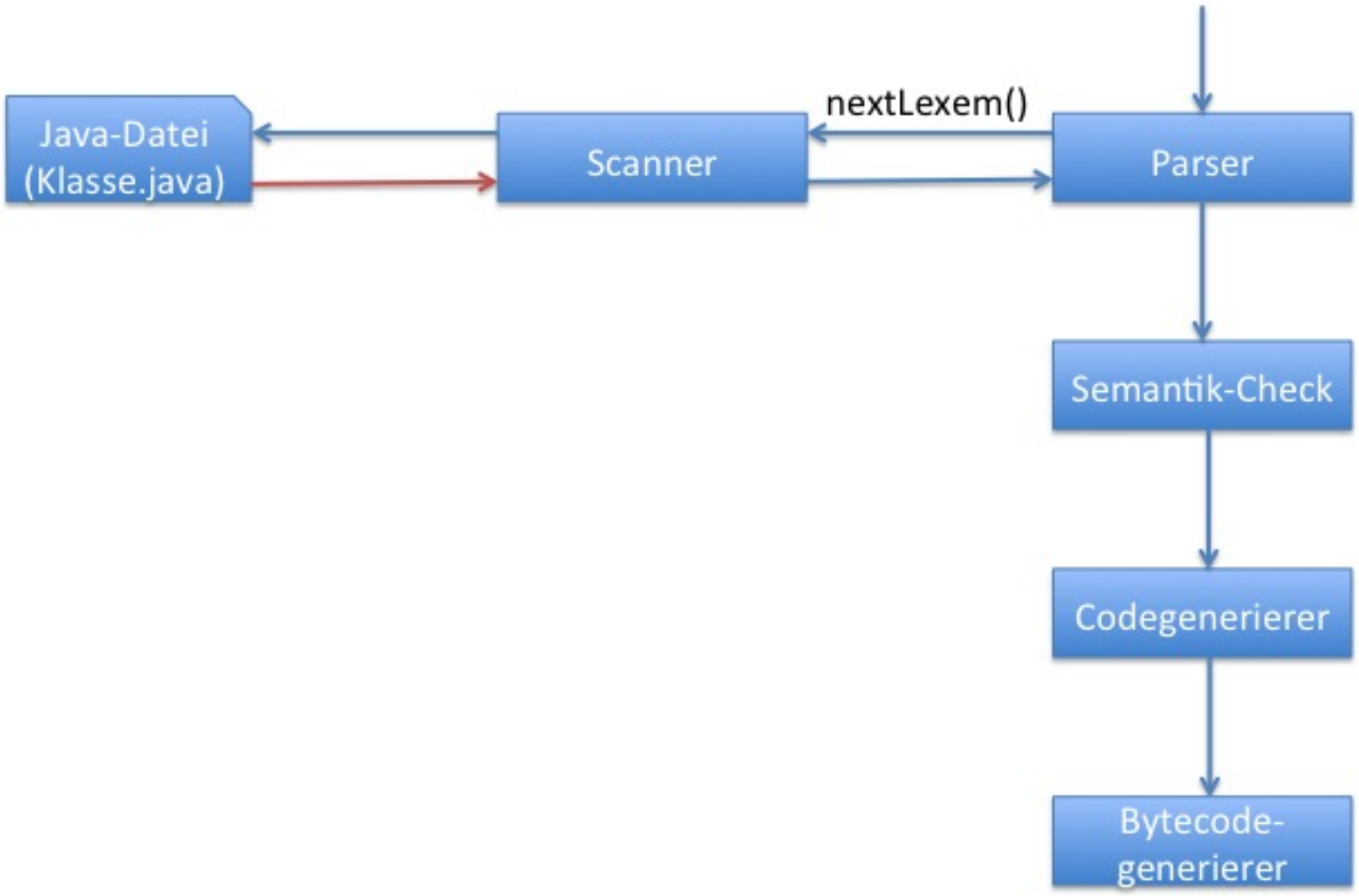

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


Peter Thiemann.
Grundlagen der funktionalen Programmierung.
Teubner, 1994.

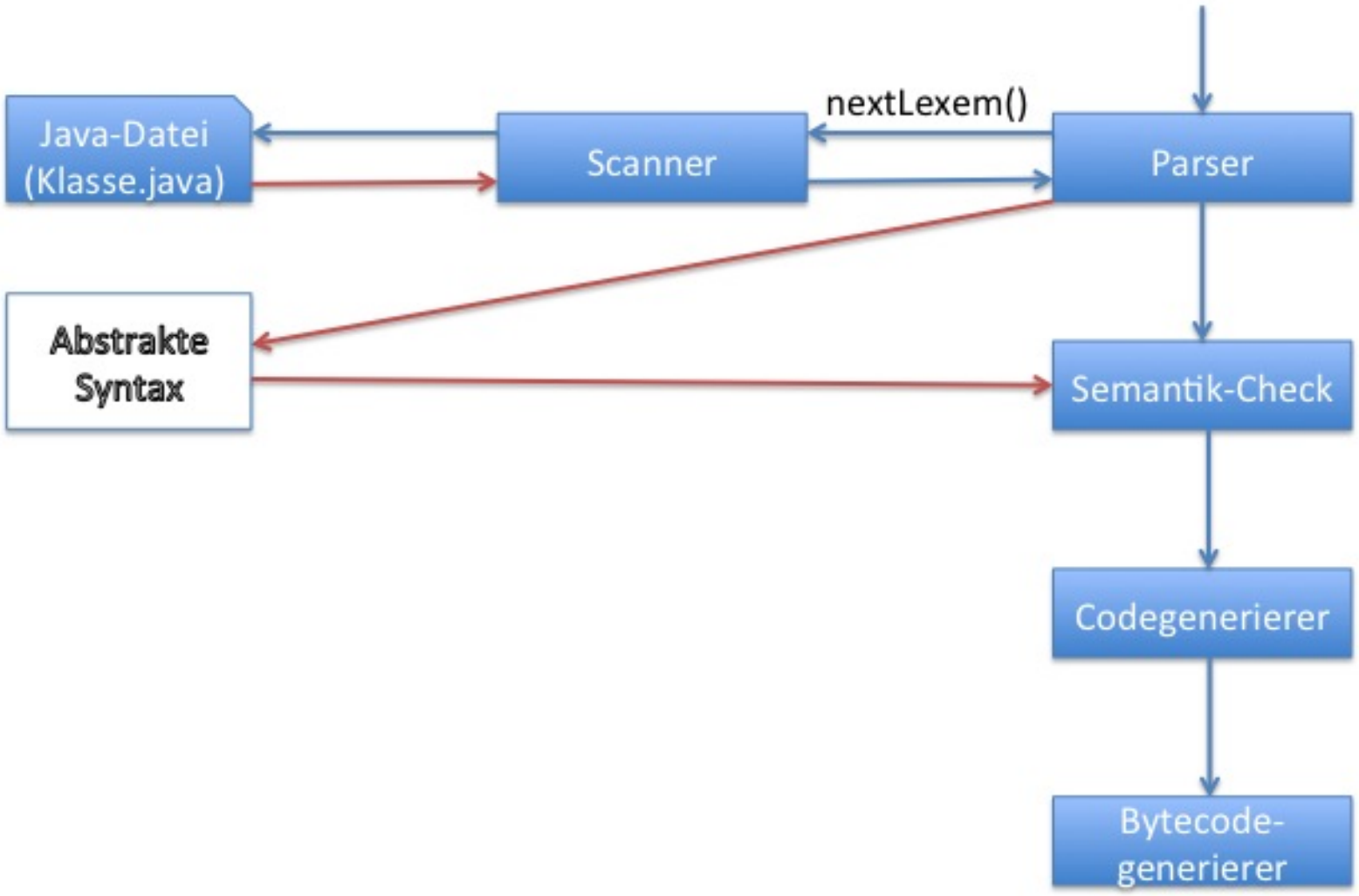
Compiler Überblick



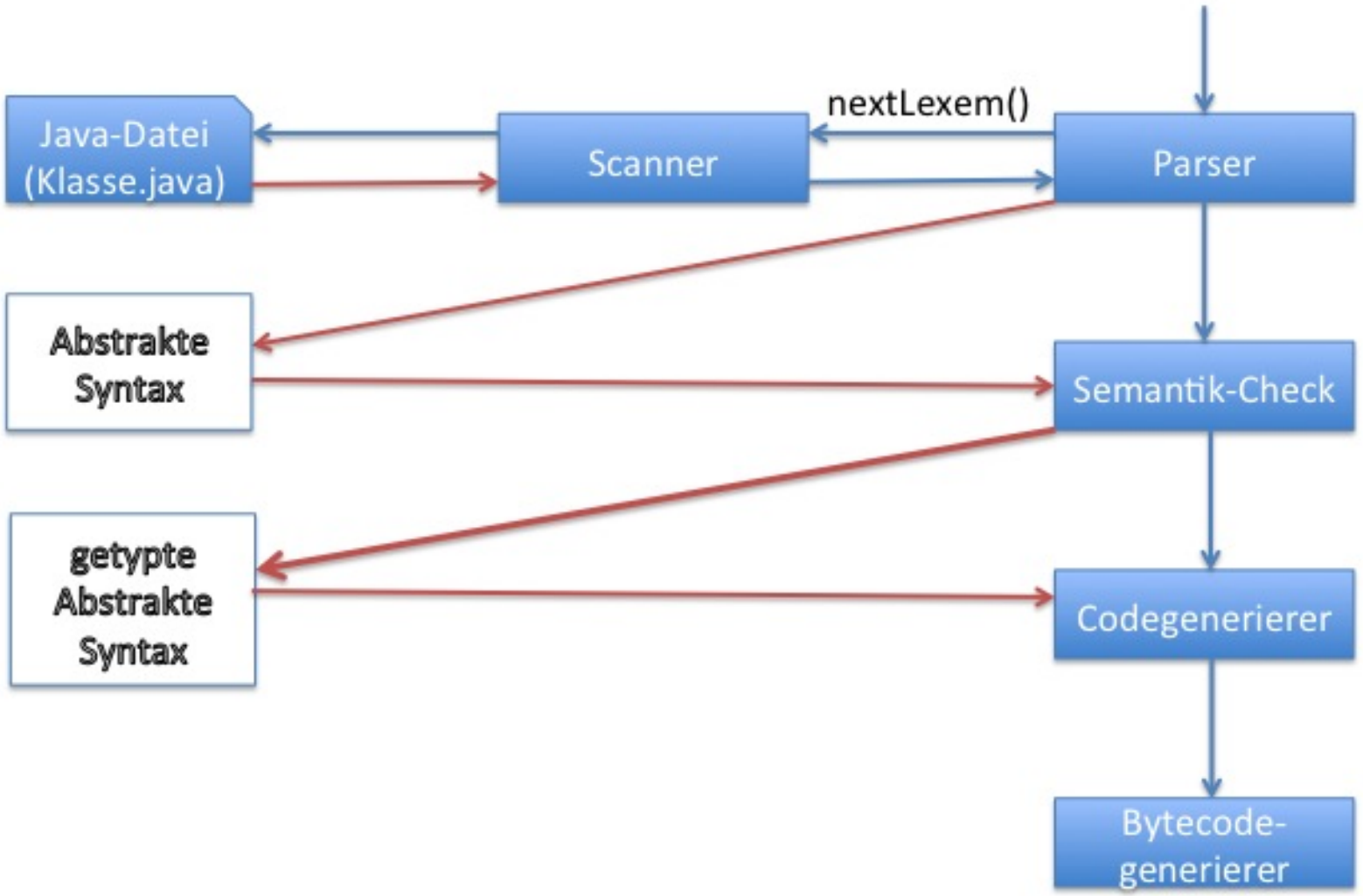
Compiler Überblick



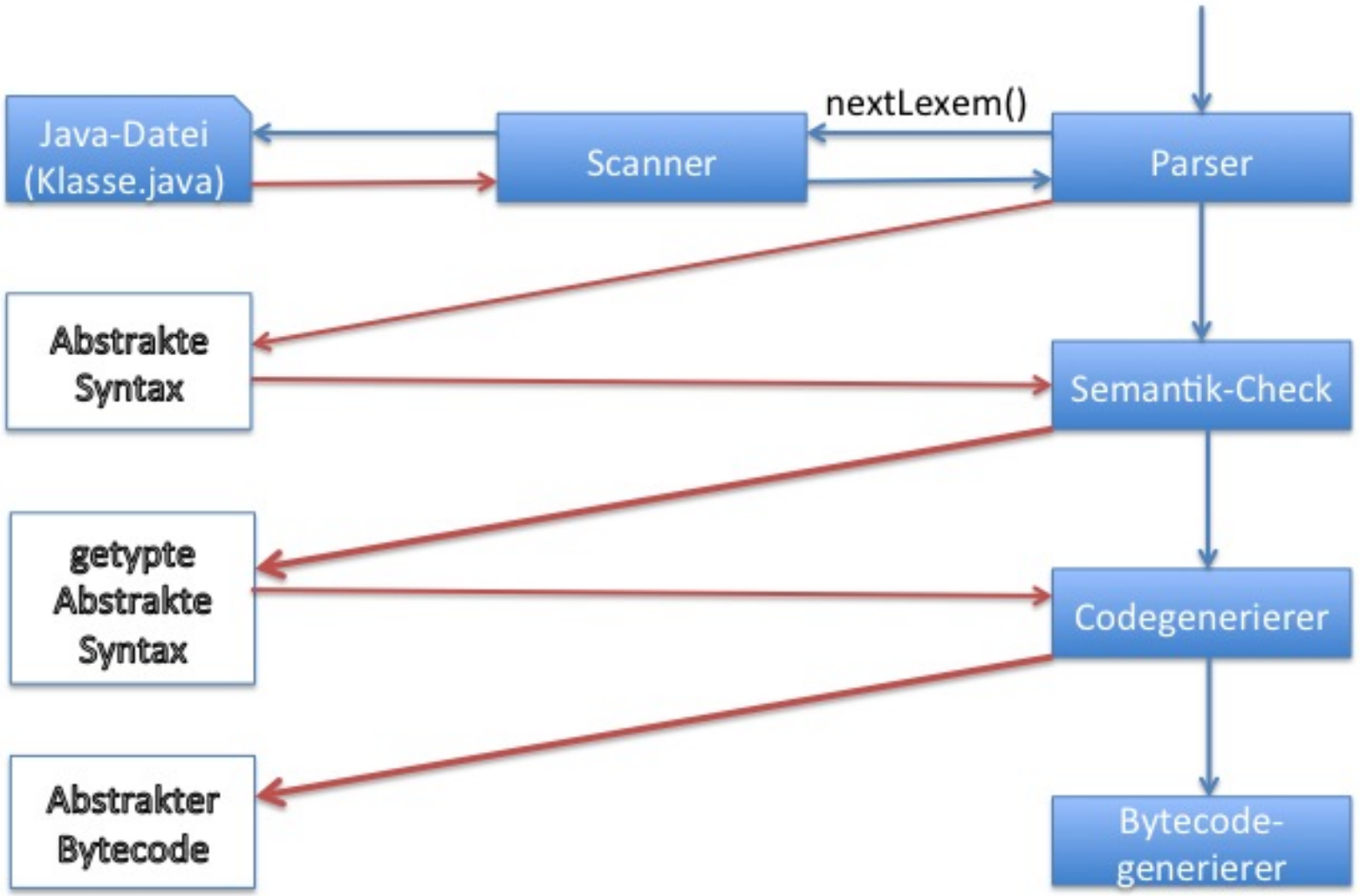
Compiler Überblick



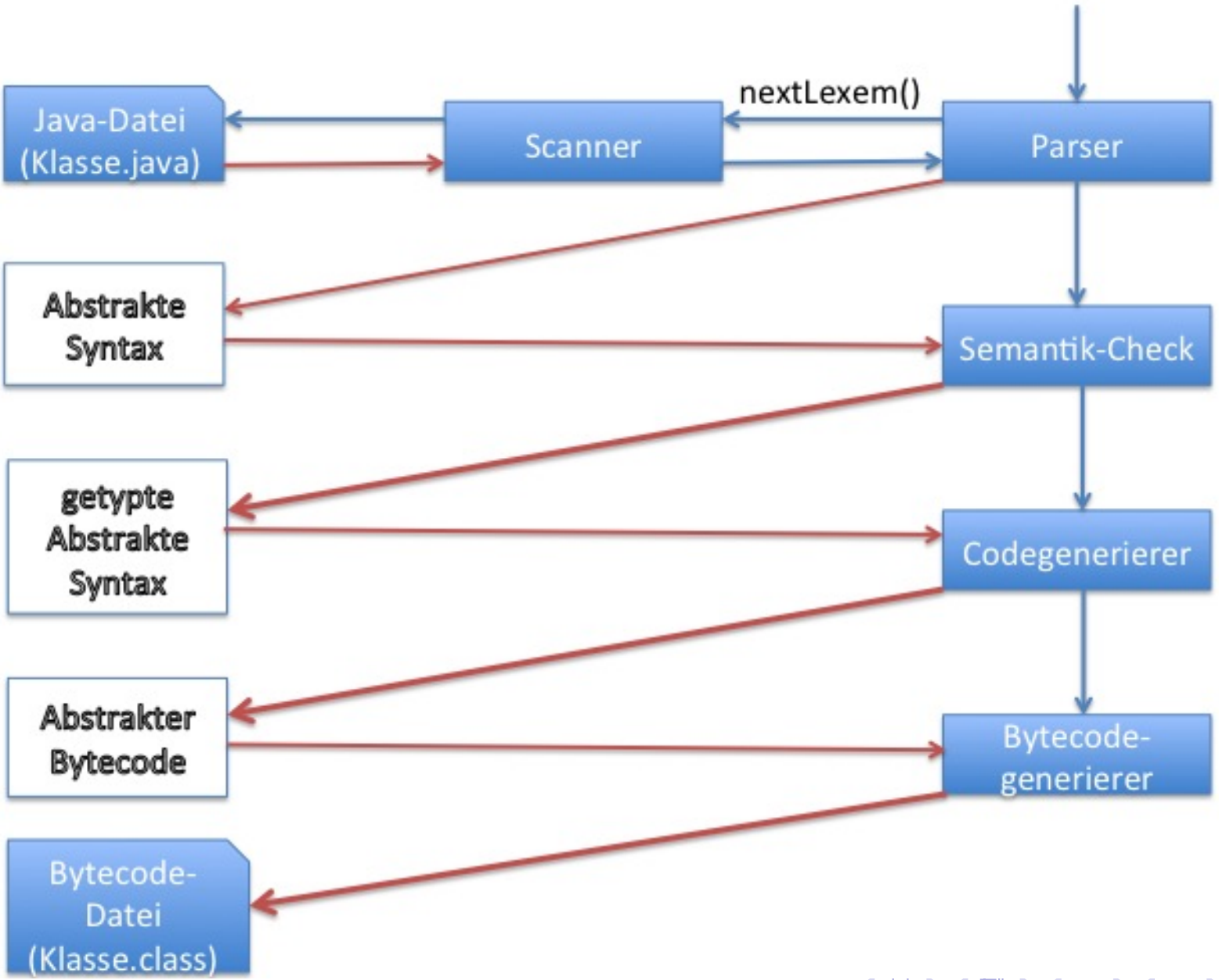
Compiler Überblick



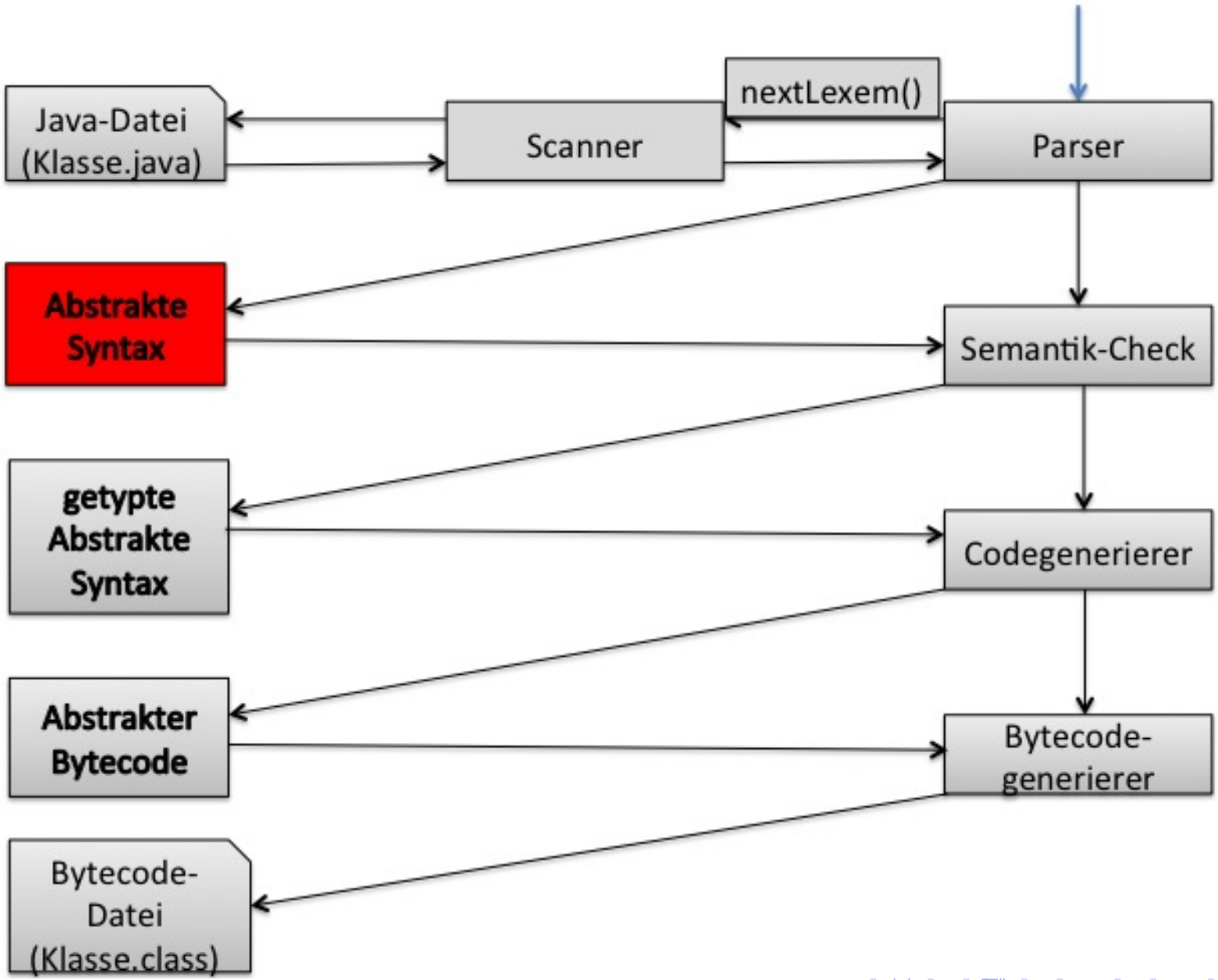
Compiler Überblick



Compiler Überblick



Abstrakte Syntax



Abstrakte Syntax

Unter abstrakter Syntax versteht man eine *abstrakte* Repräsentation eines konkreten Programms als Syntaxbaum.

Man kann den Ableitungsbaum in den abstrakten Syntaxbaum abbilden.

Abstrakte Syntax für Java
Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Abstrakte Syntax für Java
Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Java–Programm

```
class Klassenname { }
```

Abstrakte Syntax für Java
Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Java–Programm

```
class Klassenname { }
```

Abstrakte Syntax:

```
Class("Klassenname", [], [])
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = Field(Type, String)
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = Field(Type, String)
```

Java–Programm

```
class Klassenname {

  int v;

}
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = Field(Type, String)
```

Java–Programm

```
class Klassenname {

  int v;

}
```

Abstrakte Syntax:

```
Field("int", "v")
```

Methoden

Datentyp:

```
data MethodDecl =
  Method(Type, String,[(Type, String)], Stmt)
```

Methoden

Datentyp:

```
data MethodDecl =
  Method(Type, String,[(Type, String)], Stmt)
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) { }

}
```

Methoden

Datentyp:

```
data MethodDecl =
  Method(Type, String,[(Type, String)], Stmt)
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) { }

}
```

Abstrakte Syntax:

```
Method("void", "methode",
            [("int", "x"), ("char", "y")],
            Block([]))
```

Expression

```
data Expr = This
          | Super
          | LocalOrFieldVar(String)
          | InstVar(Expr, String)
          | Unary(String, Expr)
          | Binary(String, Expr, Expr)
          | Integer(Integer)
          | Bool(Bool)
          | Char(Char)
          | String(String)
          | Jnull
          | StmtExprExpr(StmtExpr)

data StmtExpr = Assign(Expr, Expr)
           | New(Type, [Expr])
           | MethodCall(Expr, String, [Expr])
```

LocalOrFieldVar

Datentyp:

```
LocalOrFieldVar(String)
```

LocalOrFieldVar

Datentyp:

`LocalOrFieldVar(String)`

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return x;

  }

}
```

LocalOrFieldVar

Datentyp:

```
LocalOrFieldVar(String)
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return x;

  }

}
```

Abstrakte Syntax:

```
LocalOrFieldVar("x")
```

InstVar

Datentyp:

```
InstVar(Expr,String)
```

InstVar

Datentyp:

`InstVar(Expr,String)`

Java–Programm

```
class Klassenname {

  int methode (Typ x, char y) {

      return x.v;

  }

}
```

InstVar

Datentyp:

```
InstVar(Expr,String)
```

Java–Programm

```
class Klassenname {

  int methode (Typ x, char y) {

      return x.v;

  }

}
```

Abstrakte Syntax:

```
InstVar(LocalOrFieldVar("x"), "v")
```

Integer

Datentyp:

```
Integer(Integer)
```

Integer

Datentyp:

`Integer(Integer)`

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return 1;

  }

}
```

Integer

Datentyp:

```
Integer(Integer)
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return 1;

  }

}
```

Abstrakte Syntax:

```
Integer(1)
```

Binary I

Datentyp:

```
Binary(String, Expr, Expr)
```

Binary I

Datentyp:

```
Binary(String, Expr, Expr)
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return 1 + x;

  }

}
```

Binary I

Datentyp:

```
Binary(String, Expr, Expr)
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {

      return 1 + x;

  }

}
```

Abstrakte Syntax:

```
Binary("+", Integer(1), LocalOrFieldVar("x"))
```

Binary II

Datentyp:

```
Binary(String, Expr, Expr)
```

Binary II

Datentyp:

```
Binary(String, Expr, Expr)
```

Java–Programm

```
class Klassenname {
  void methode (int x, int y) {

      if (x == y) { }

  }
}
```

Binary II

Datentyp:

```
Binary(String, Expr, Expr)
```

Java–Programm

```
class Klassenname {
  void methode (int x, int y) {

      if (x == y) { }

  }
}
```

Abstrakte Syntax:

```
Binary("==",
          LocalOrFieldVar("x"),
          LocalOrFieldVar("y"))
```

Statement Expression: MethodCall

Datentyp:

```
StmtExprExpr(StmtExpr)
MethodCall(Expr,String,[Expr])
```

Statement Expression: MethodCall

Datentyp:

```
StmtExprExpr(StmtExpr)
MethodCall(Expr,String,[Expr])
```

Java–Programm

```
class Klassenname {

  int methode (Typ x, int y, int z) {
      return x.f(y, z);
  }
}
```

Statement Expression: MethodCall

Datentyp:

```
StmtExprExpr(StmtExpr)
MethodCall(Expr,String,[Expr])
```

Java–Programm

```
class Klassenname {

  int methode (Typ x, int y, int z) {
      return x.f(y, z);
  }
}
```

Abstrakte Syntax:

```
StmtExprExpr(MethodCall(LocalOrFieldVar("x")),
             "f",
             [LocalOrFieldVar("y"), LocalOrFieldVar("z")])
```

Statements

```
data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl( Type, String )
          | If( Expr, Stmt , Maybe Stmt )
          | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(Expr, Expr)
          | New(Type, [Expr])
          | MethodCall(Expr, String, [Expr])
```

Return–Statement

Datentyp:

```
Return( Expr )
```

Return–Statement

Datentyp:

```
Return( Expr )
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {
      return 1 + x;
  }

}
```

Return–Statement

Datentyp:

```
Return( Expr )
```

Java–Programm

```
class Klassenname {

  int methode (int x, char y) {
      return 1 + x;
  }

}
```

Abstrakte Syntax:

```
Return(Binary("+",
              Integer(1),
              LocalOrFieldVar("x")))
```

While–Statement

Datentyp:

```
While( Expr, Stmt )
```

While–Statement

Datentyp:

```
While( Expr, Stmt )
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) {
      while (x < 1) { }
  }

}
```

While–Statement

Datentyp:

```
While( Expr, Stmt )
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) {
      while (x < 1) { }
  }

}
```

Abstrakte Syntax:

```
While(Binary("<",
                  LocalOrFieldVar("x"),
                  Integer(1)),
      Block([]))
```

LocalVarDecl–Statement

Datentyp:

```
LocalVarDecl( Type, String )
```

LocalVarDecl–Statement

Datentyp:

```
LocalVarDecl( Type, String )
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) {
      int i;
  }

}
```

LocalVarDecl–Statement

Datentyp:

```
LocalVarDecl( Type, String )
```

Java–Programm

```
class Klassenname {

  void methode (int x, char y) {
      int i;
  }

}
```

Abstrakte Syntax:

```
LocalVarDecl("int", "i")
```

If (ohne else)

Datentyp:

```
If(Expr, Stmt, Maybe Stmt)
```

If (ohne else)

Datentyp:

```
If(Expr, Stmt, Maybe Stmt)
```

Java–Programm

```
class Klassenname {
  int methode (int x, int y) {
      if (x == y)  return 1;

  }
```

If (ohne else)

Datentyp:

```
If(Expr, Stmt, Maybe Stmt)
```

Java–Programm

```
class Klassenname {
  int methode (int x, int y) {
      if (x == y)  return 1;

}
```

Abstrakte Syntax:

```
If(Binary("==",
                LocalOrFieldVar("x"),
                LocalOrFieldVar("y")),
   Return(Integer(1))),
   Nothing)
```

If (mit else)

Datentyp:

```
If(Expr, Stmt, Maybe Stmt)
```

Java–Programm

```
class Klassenname {
  int methode (int x, int y) {
      if (x == y)  return 1;
      else return 2;
}
```

Abstrakte Syntax:

```
If(Binary("==",
                LocalOrFieldVar("x"),
                LocalOrFieldVar("y")),
   Return(Integer(1)),
   Just (Return(Integer(2))))
```

Statement Expression: Assign

Datentyp:

```
StmtExprStmt(StmtExpr)
Assign(Expr, Expr)
```

Statement Expression: Assign

Datentyp:

```
StmtExprStmt(StmtExpr)
Assign(Expr, Expr)
```

Java–Programm

```
class Klassenname {

  void methode (Typ x, int y, int z) {
      int i;
      i = x;
  }
}
```

Statement Expression: Assign

Datentyp:

```
StmtExprStmt(StmtExpr)
Assign(Expr, Expr)
```

Java–Programm

```
class Klassenname {

  void methode (Typ x, int y, int z) {
      int i;
      i = x;
  }
}
```

Abstrakte Syntax:

```
StmtExprStmt(Assign(LocalOrFieldVar("i"),
                    LocalOrFieldVar("x")))
```

Komplettes Beispiel

```
class Klassenname {
  int v;
  int methode (Typ x, int y, int z) {
      int i;
      i = v;
      return i;
   }
}
```

Komplettes Beispiel

```
class Klassenname {
  int v;
  int methode (Typ x, int y, int z) {
      int i;
      i = v;
      return i;
   }
}
```

Abstrakte Syntax:

```
Class("Klassenname",
      [Field("int", "v")],
      [Method ("int", "methode",
         [("Typ", "x"), ("int", "y"), ("int", "z")],
         Block([LocalVarDecl("int, "i"),
                StmtExprStmt(
                  Assign(LocalOrFieldVar("i"),
                         LocalOrFieldVar("v"))),
                Return (LocalOrFieldVar(i))]))])
```

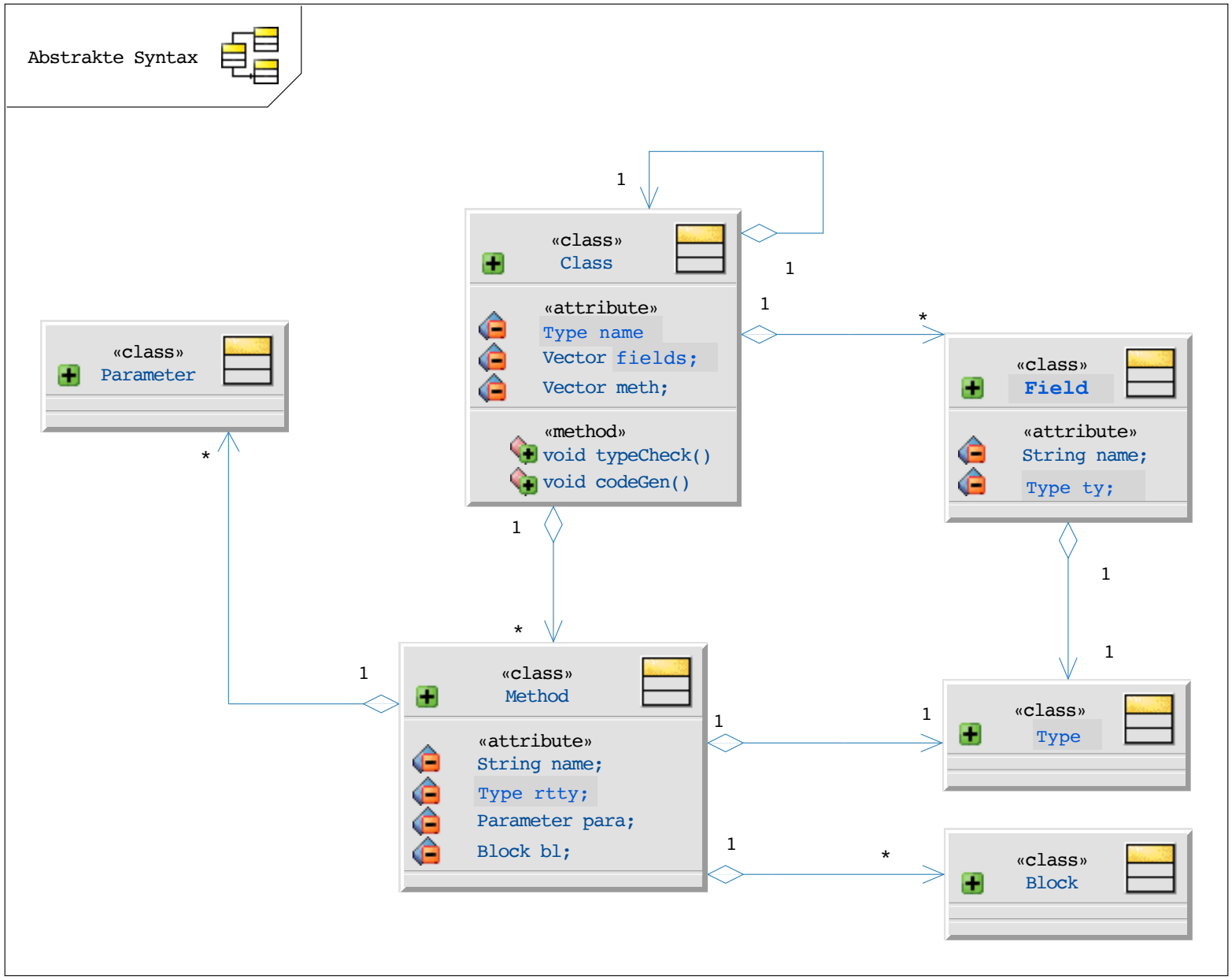
Ungetypte abstrakte Syntax für *Mini*–Java I

```
data Class = Class(Type, [FieldDecl], [MethodDecl])

data FieldDecl = Field(Type, String)

data MethodDecl = Method(Type, String,[(Type,String)], Stmt)

data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl(Type, String)
          | If(Expr, Stmt , Maybe Stmt)
          | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(String, Expr)
              | New(Type, [Expr])
              | MethodCall(Expr, String, [Expr])
```

Ungetypte abstrakte Syntax für *Mini*–Java II

```
data Expr = This
          | Super
          | LocalOrFieldVar(String)
          | InstVar(Expr, String)
          | Unary(String, Expr)
          | Binary(String, Expr, Expr)
          | Integer(Integer)
          | Bool(Bool)
          | Char(Char)
          | String(String)
          | Jnull
          | StmtExprExpr(StmtExpr)

type Prg = [Class]
```

Übersetzung in eine Java Datenstruktur



Übersetzung Klasse

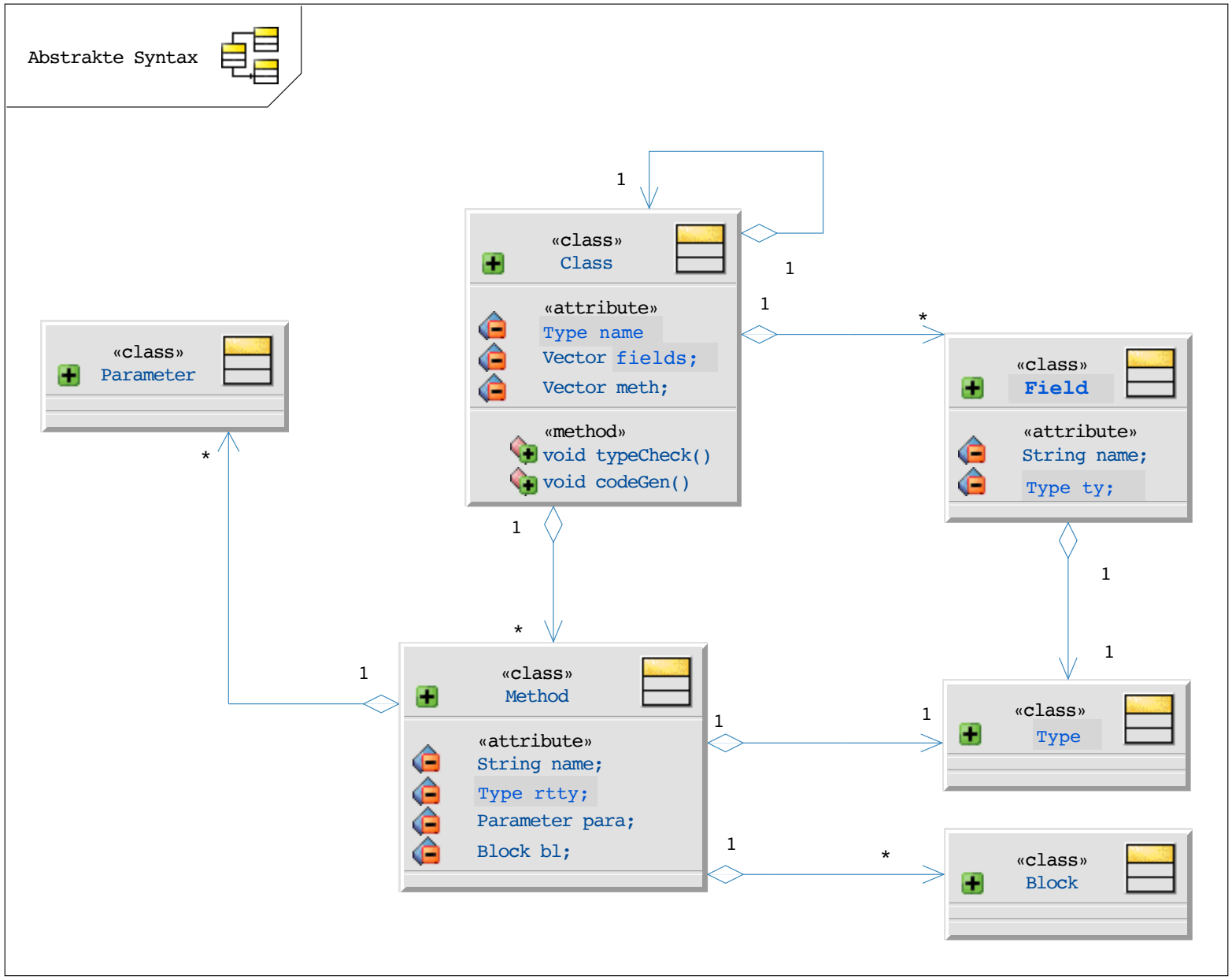
```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Übersetzung Klasse

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

```
class Class {
  Type name
  Vector<Field> fields;
  Vector<Method> meths;
}
```

Übersetzung in eine Java Datenstruktur



Übersetzung Field

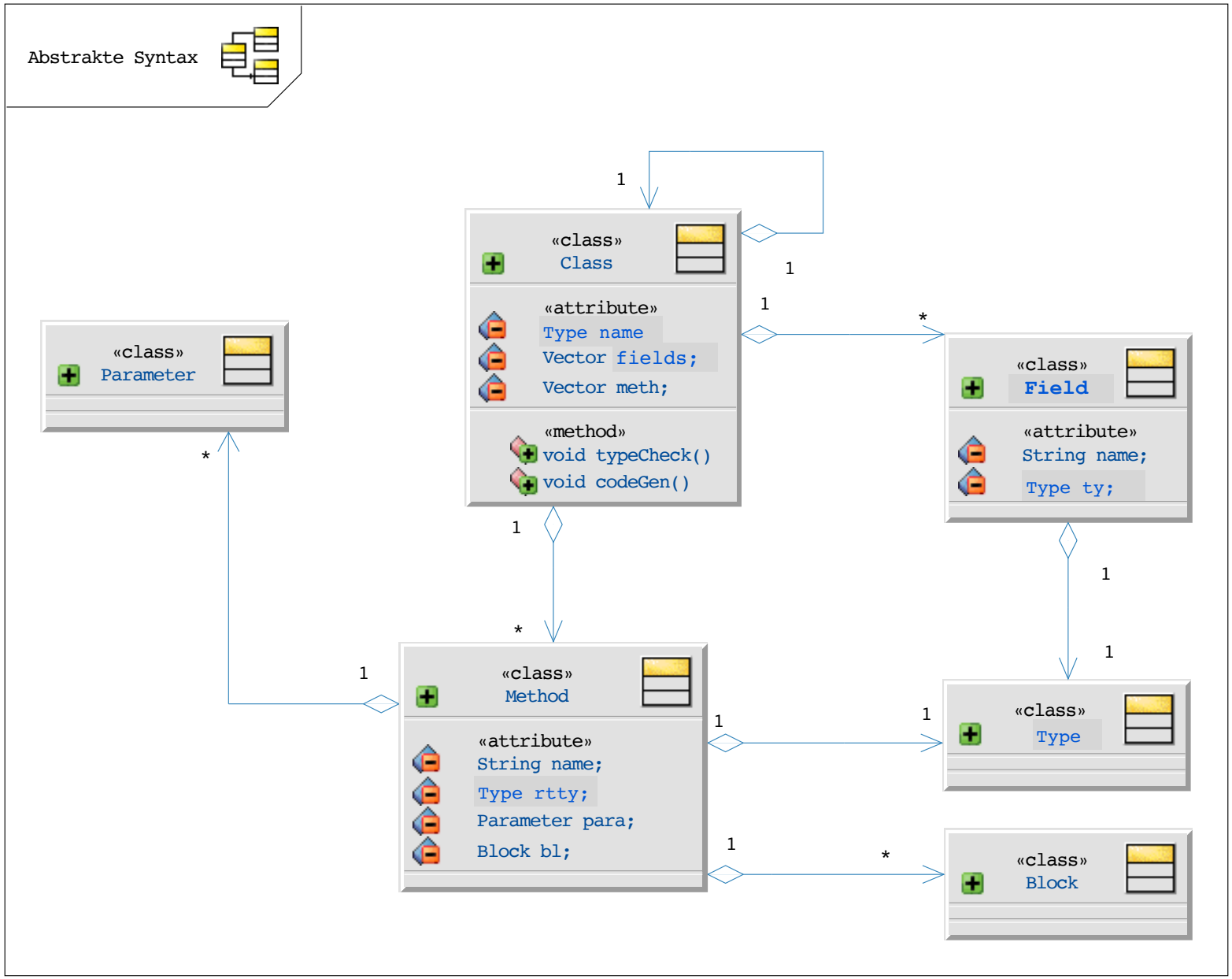
```
data FieldDecl = Field(Type, String)
```

Übersetzung Field

```
data FieldDecl = Field(Type, String)
```

```
class Field {
  String name;
  Type ty;
}
```

Übersetzung in eine Java Datenstruktur



Übersetzung Method

```
data MethodDecl = Method(Type, String,[(Type,String)], Stmt)
```

Übersetzung Method

```
data MethodDecl = Method(Type, String,[(Type,String)], Stmt)
```

```
class Method {
    String name;
    Type retty;
    Parameter para;
    Block bl;
}
```

Parser/Transformation konkrete -> abstrakte Syntax

Parsergenerator Happy

- Yacc für Haskell
- `AbsSyn.hs`, `JavaLexer.hs` und `JavaParser.y`

  erzeugt

  `JavaParser.hs`
- **Aufruf:** `> happy -info JavaParser.y`
- Option `-info` erzeugt die Info-Datei: `nfo`

Das Happy-File
I. Haskell–Header

Haskell–Source–Code:

```
{
module JavaParser (parse) where
import AbsSyn
}
```

- Das erzeugte Haskell–File definiert das Module `JavaParser`.
- Die Funktion `parse` wird exportiert.
- Das Modul `AbsSyn` wird importiert.

II. Deklarationen

```
name parse
tokentype  Token
error  parseError
```

- `name`: Name der Parserfunktion.
- `tokentype`: Type der einzelnen Tokens, die der Parser liest.
- `error`: Name der Funktion, die bei einem Fehler aufgerufen wird.

III. Tokens

```
token
        BOOLEAN { BOOLEAN }
        BREAK { BREAK }
        CASE { CASE }
        CHAR  { CHAR  }
        CLASS { CLASS}
        IDENTIFIER { IDENTIFIER $$ }
        INTLITERAL { INTLITERAL $$ }
        ...
```

- Die Tokens werden definiert durch das Paar
  - *Terminal* in der Grammatik (links)
  - *Haskell-Konstruktor des Typs* `tokentype` (rechts in geschweifter Klammer)
- Wert des Tokens: normalerweise das Token selbst
  $$ bedeutet, der Wert ist das Argument des Tokens

Data–Deklaration des tokentype

Am Ende der happy–Datei, oder in importierter Haskell-Datei (hier `JavaLexer.hs`):

```
data Token = BOOLEAN
           | BREAK
           | CASE
           | CHAR
           | CLASS
           | IDENTIFIER String
           | INTLITERAL Int
           ...
```

Übersetzung Ableitungsbaum -> abstrakter Syntaxbaum

```
%% -- aus yacc-Tradition

classdeclaration : CLASS IDENTIFIER classbody
                   { Class($2, fst $3, snd $3 ) }
                 | modifiers CLASS IDENTIFIER classbody
                   { Class($3, fst $4, snd $4) }

classbody        : LBRACKET RBRACKET  { ([], []) }
                 | LBRACKET classbodydeclarations  RBRACKET { $2 }

modifiers        : modifier { }
                 | modifiers modifier { }
```

- Hinter jeder Regel wird eine Haskell–Anweisung angegeben, die beim *Reduce*–Schritt des Parsers ausgeführt wird.
- $n gibt das Ergebnis des n. Symbols der rechten Seite an.

Beispiel

```
CLASS IDENTIFIER classbody { Class($2, fst $3, snd $3 ) }
```

bedeutet: Beim *reduce* wird ein `Class`–Element erzeugt, das als

1. Argument den *IDENTIFIER* ($2) und als
2. Argument first des *classbody*s ($3) und als
3. Argument second des *classbody*s ($3)

hat.

Beispiel

```
CLASS IDENTIFIER classbody { Class($2, fst $3, snd $3 ) }
```

bedeutet: Beim *reduce* wird ein `Class`–Element erzeugt, das als

1. Argument den *IDENTIFIER* ($2) und als
2. Argument first des *classbody*s ($3) und als
3. Argument second des *classbody*s ($3)

hat.

```
classbody : LBRACKET RBRACKET { ([], []) }
```

gibt das Paar `([], [])` zurück.

Beispiel

```
CLASS IDENTIFIER classbody { Class($2, fst $3, snd $3 ) }
```

bedeutet: Beim *reduce* wird ein `Class`–Element erzeugt, das als

1. Argument den *IDENTIFIER* ($2) und als
2. Argument first des *classbody*s ($3) und als
3. Argument second des *classbody*s ($3)

hat.

```
classbody : LBRACKET RBRACKET { ([], []) }
```

gibt das Paar `([], [])` zurück.

```
fst ([], []) = []
snd ([], []) = []
```