Compilerbau

Martin Plümicke
Andreas Stadelmeier

SS 2024

Beschreibung

In der Vorlesung werden anwendungsnahe Konzepte und Techniken zu Programmiersprachen und Compilerbau vermittelt. Konkret werden zunächst die Phasen des Compilerbaus an Hand eines Java-Compilers vorgestellt. Als Implementierungstechnik wird die funktionale Programmiersprache Haskell verwendet. Dazu werden die notwendigen Grundlagen der funktionalen Programmierung aufbauend auf den Kenntissen der Grundvorlesung vermittelt. Im 2. Teil der Lehrveranstaltungen werden die Studierenden in Gruppenarbeit einen Mini-Java-Compiler mit den gelernten Techniken implementieren.

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

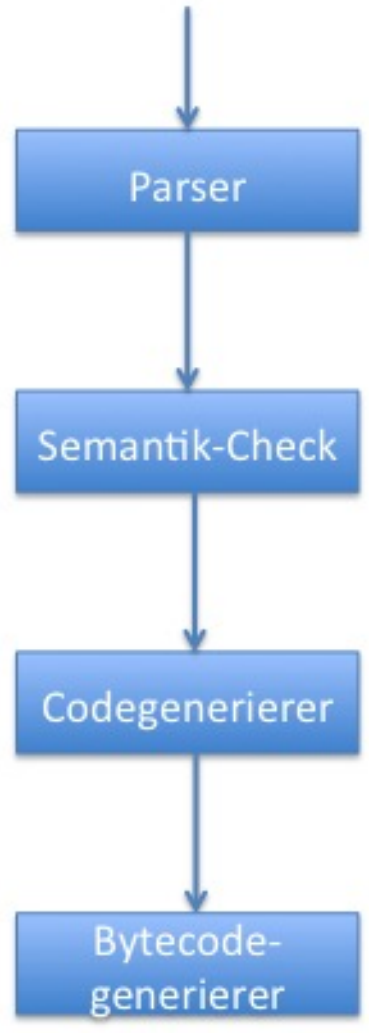

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


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

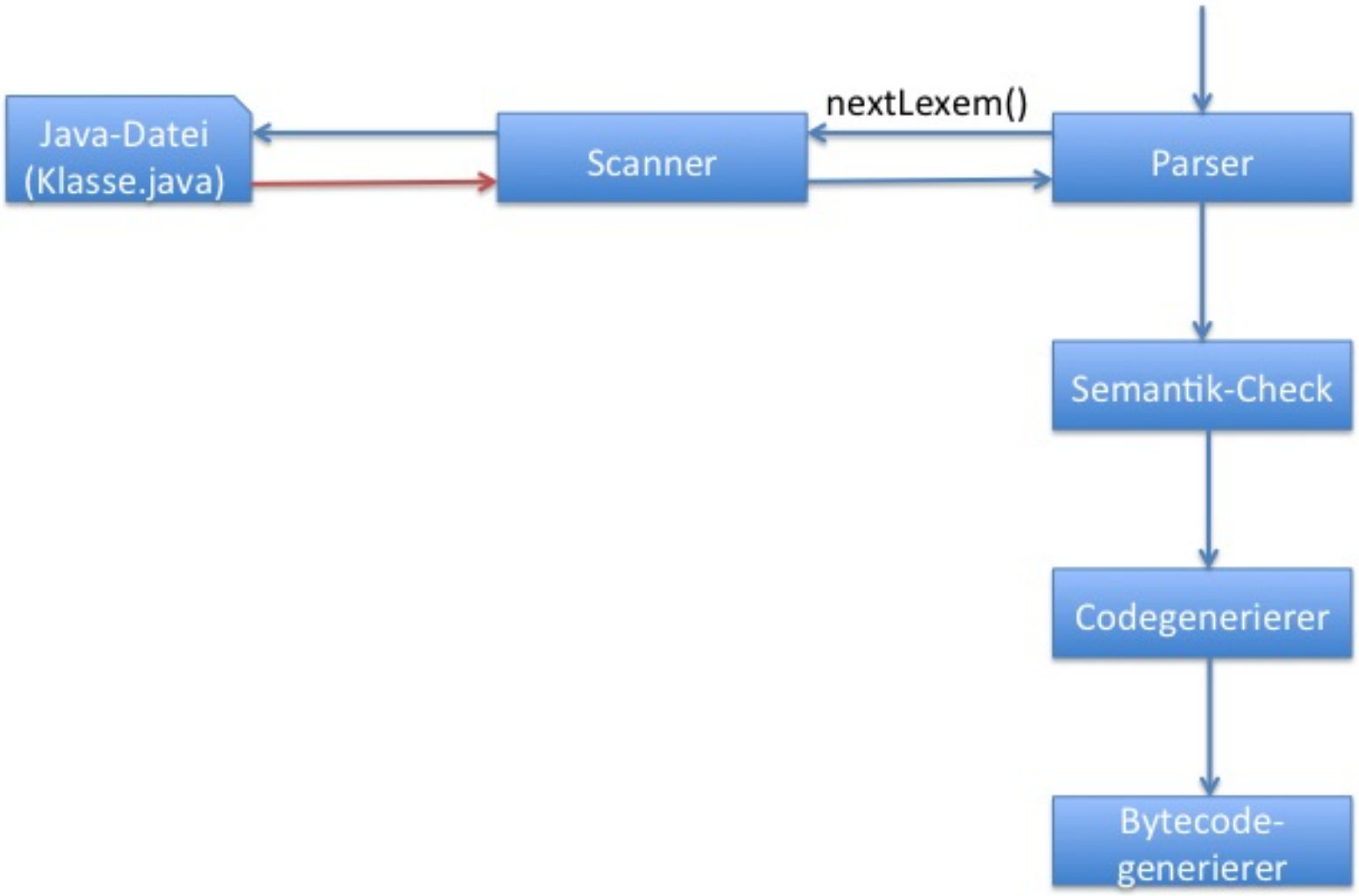

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


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

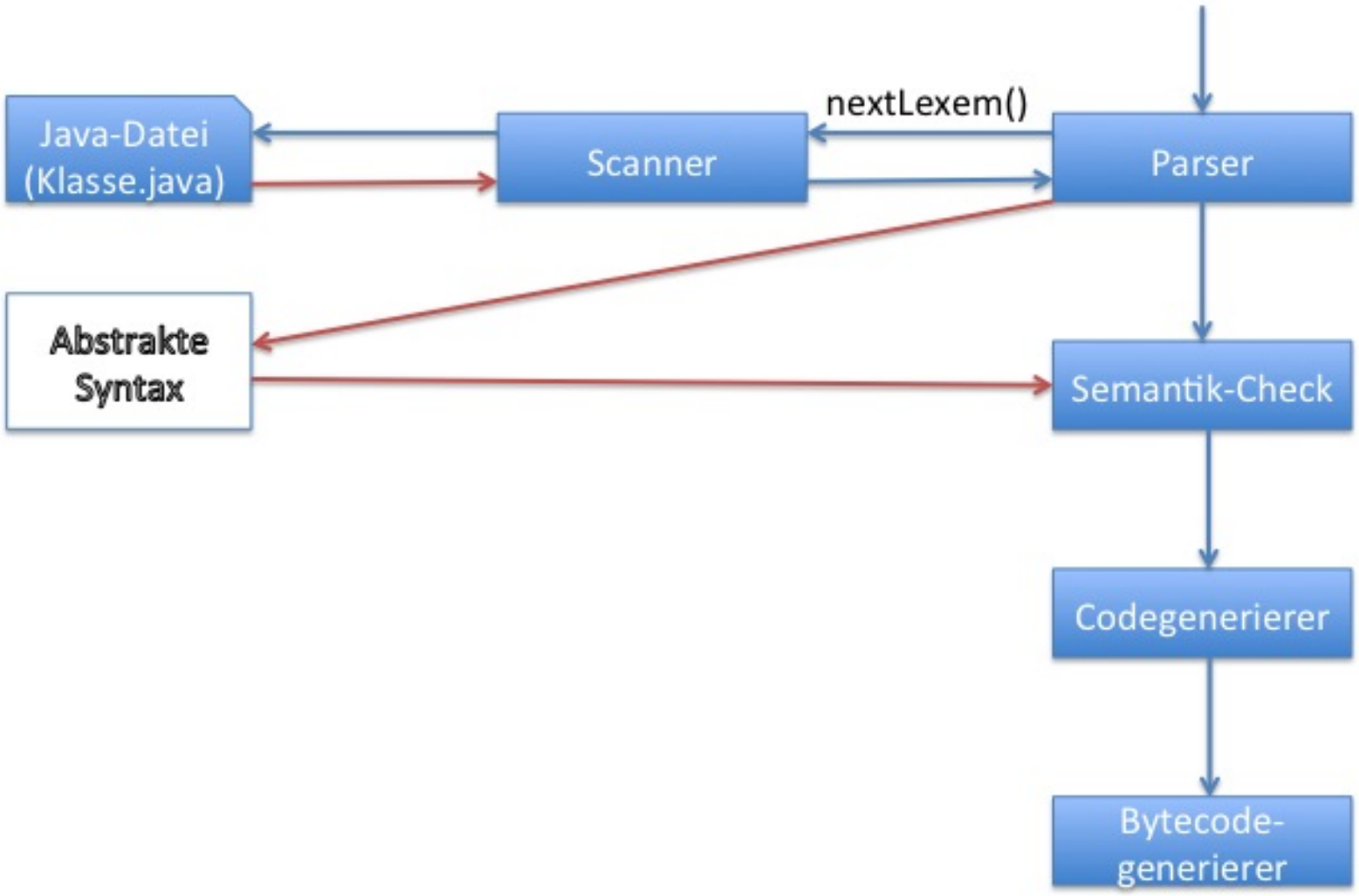
Compiler Überblick



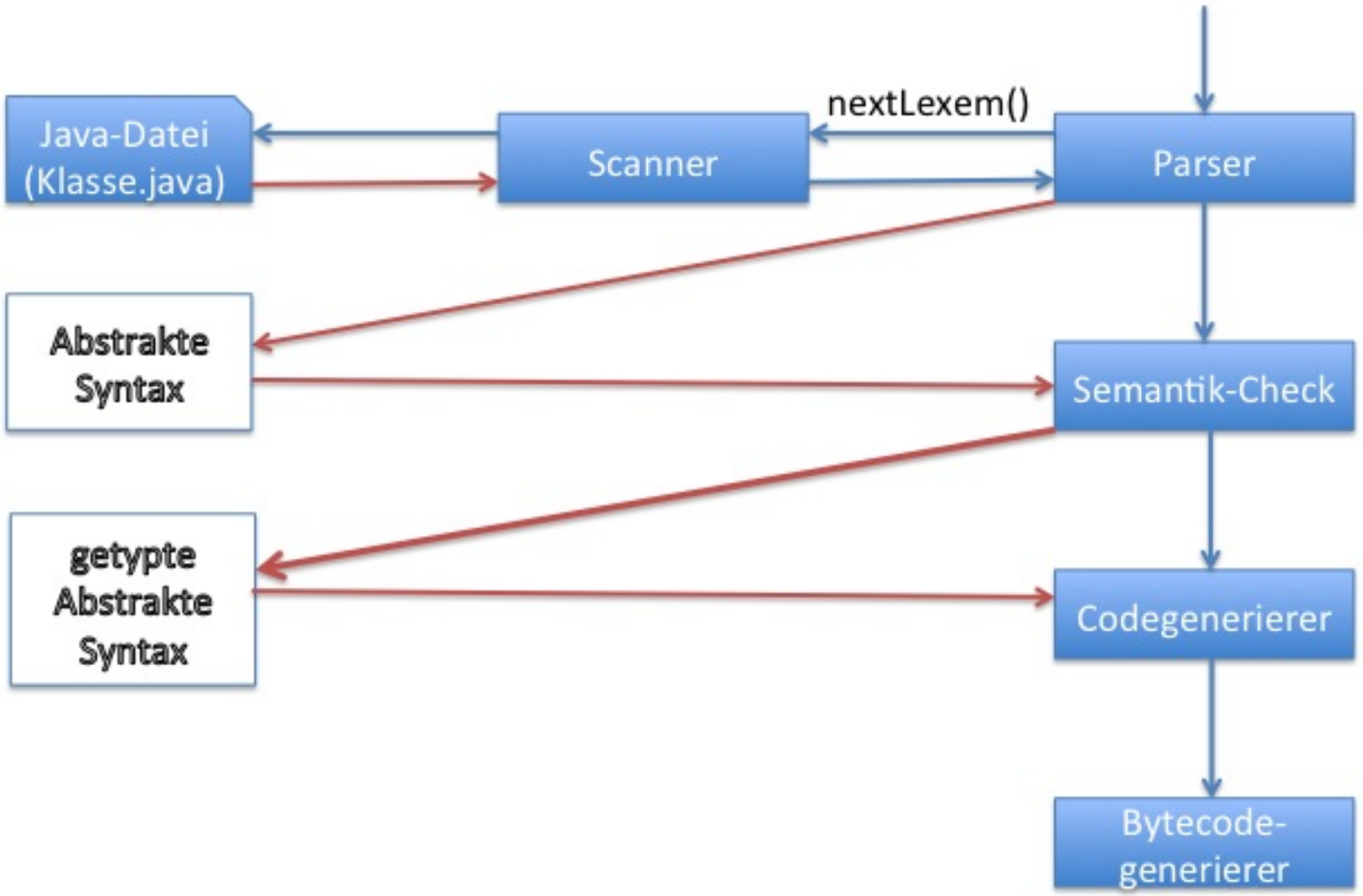
Compiler Überblick



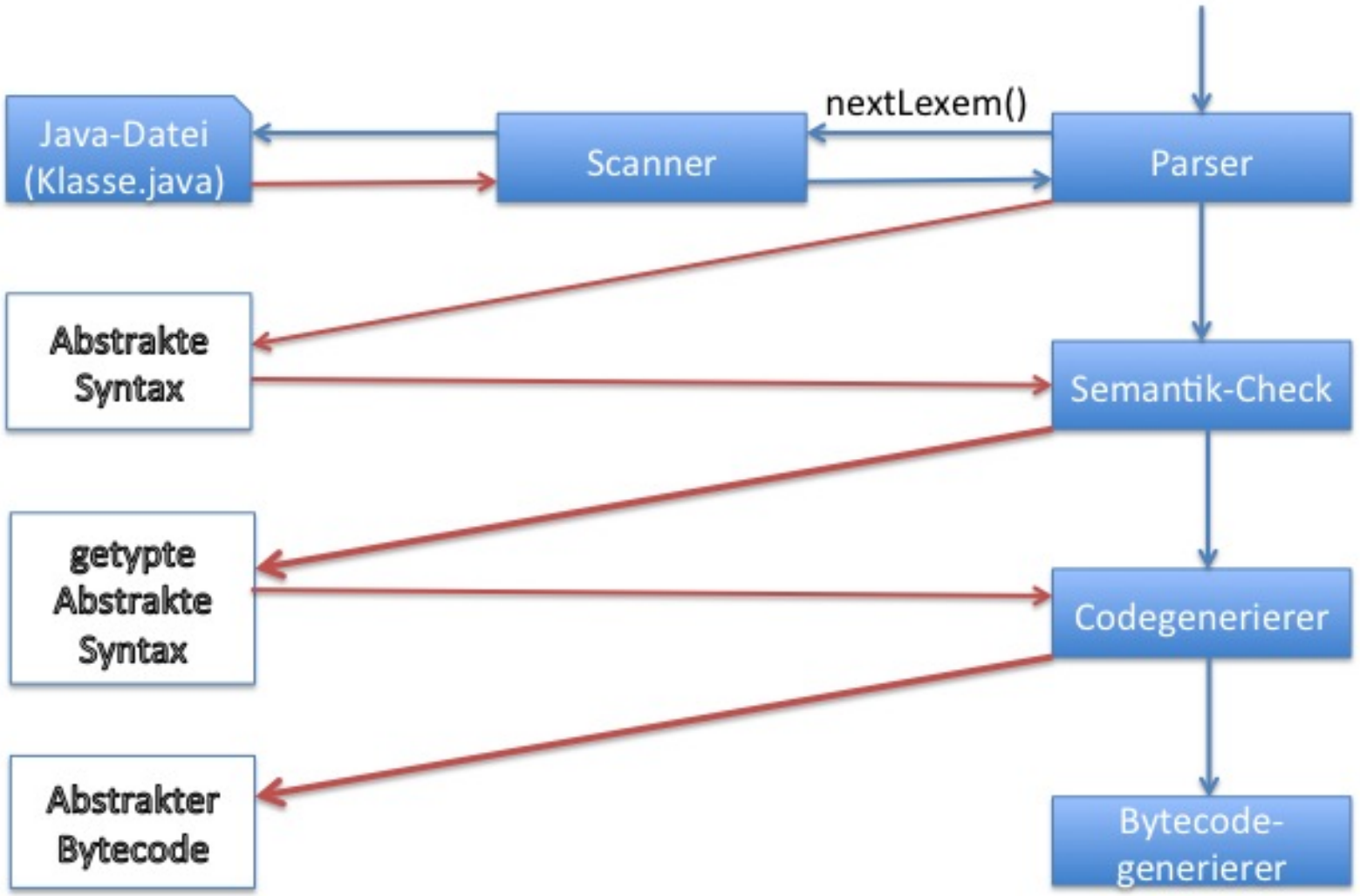
Compiler Überblick



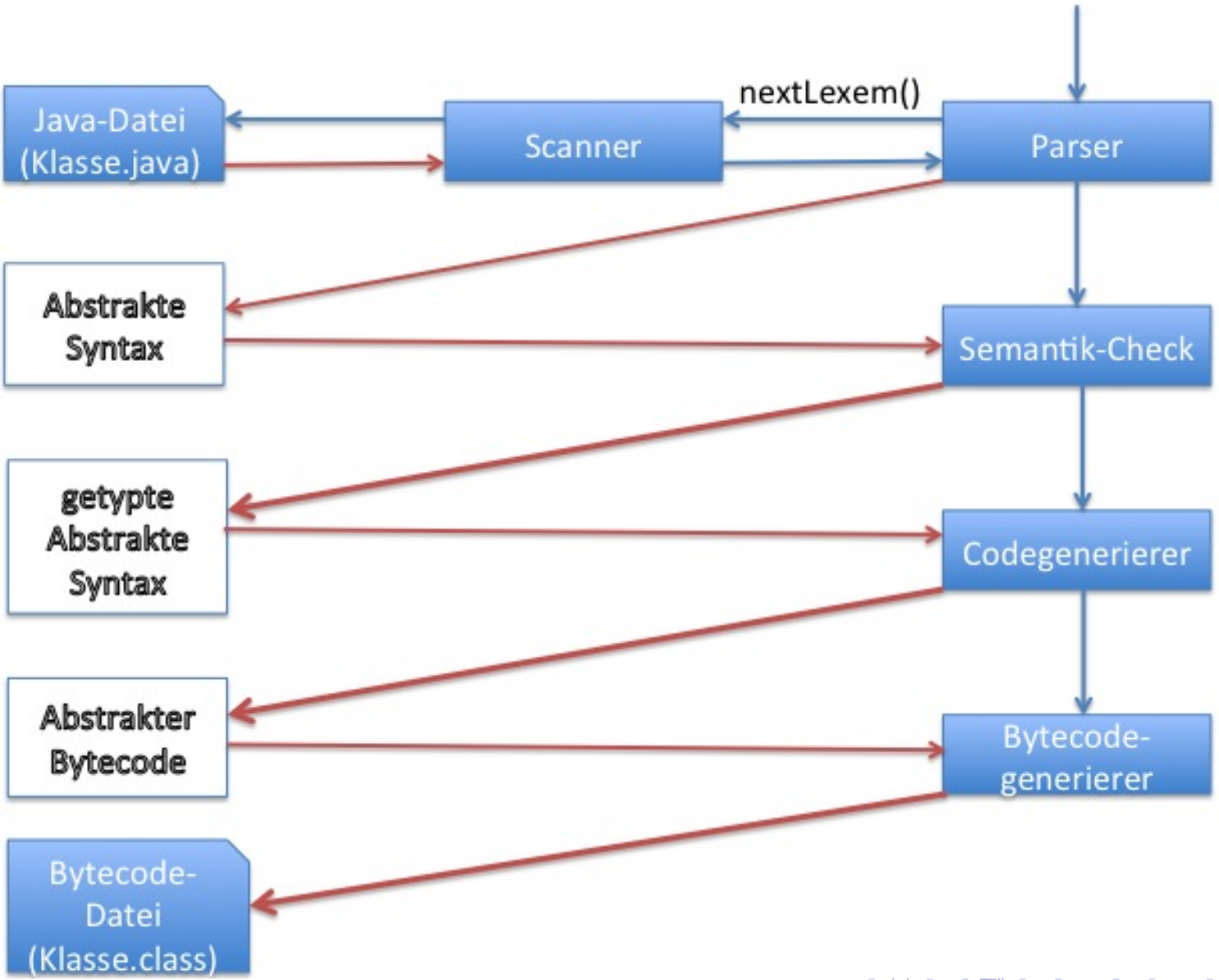
Compiler Überblick



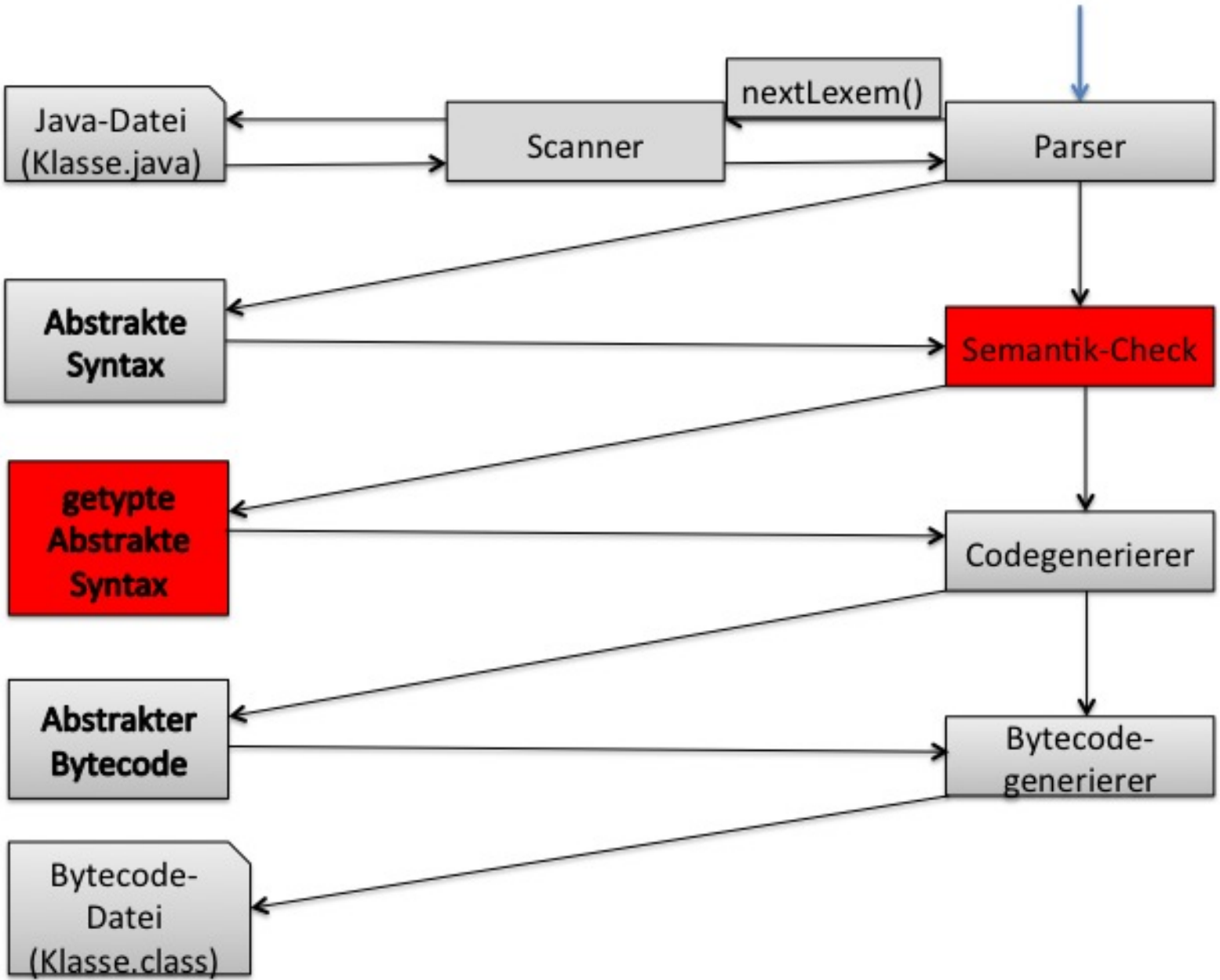
Compiler Überblick



Compiler Überblick



Semantische Analyse/Typecheck



Semantische Analyse/Typecheck

- Überprüfen der Kontextsensitiven Nebenbedingungen:
  - alle Variablen/Methoden deklariert/sichtbar?
  - Typen korrekt?
- Typisierung aller Sub–Terme

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]
```

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

$$O \rhd_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

$$O \rhd_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \rhd_{Expr} e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

$$O \rhd_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \rhd_{Expr} e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

$$O \rhd_{Stmt} s : ty$$

Aus der Menge der Typannahmen O ist für das Statement s der Typ ty ableitbar.

Formale Definitionen II

Typurteile

$$[\textbf{Regelname}]\ \frac{O1 \rhd x : ty1}{O2 \rhd y : ty2}$$

Aus der Regel *Regelname* folgt, wenn man aus $O1$ ableiten kann, dass x den Typ ty1 hat, dann kann man aus $O2$ ableiten dass y den Typ $ty2$ hat.

Ident–Rule

$$[\textbf{Ident}]\ \frac{(f : ty) \in O_\tau}{O_\tau \ \rhd_{Id}\ f : ty}$$

O_τ: Menge aller in der Klasse τ sichtbaren Methoden und Attribute

Beispiel Ident–Rule

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

Beispiel Ident–Rule

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

- $O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id} \mathtt{attr} : \mathtt{int}}$$

Beispiel Ident–Rule

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

- $O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id} \mathtt{attr} : \mathtt{int}}$$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id} \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A}}$$

Literal–Regeln

$$[\textbf{IntLiteral}] \quad O \rhd_{Expr} \text{Integer}(\, n \,) : \texttt{int}$$

$$[\textbf{BoolLiteral}] \quad O \rhd_{Expr} \text{Bool}(\, b \,) : \texttt{boolean}$$

$$[\textbf{CharLiteral}] \quad O \rhd_{Expr} \text{Char}(\, c \,) : \texttt{char}$$

$$[\textbf{NullLiteral}] \quad O \rhd_{Expr} \text{Null} : \theta'$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O \ \triangleright_{Expr}\ e : \texttt{int}}{O \ \triangleright_{Expr}\ \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O \rhd_{Expr} e : \texttt{int}}{O \rhd_{Expr} \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\ \frac{O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Expr} \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O \rhd_{Expr} e : \texttt{int}}{O \rhd_{Expr} \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\ \frac{O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Expr} \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

$$[\textbf{Binary1}]\ \frac{O \rhd_{Expr} e1 : \texttt{int}, O \rhd_{Expr} e2 : \texttt{int}}{O \rhd_{Expr} \text{Binary}(\text{"+"}/\text{"-"}/\text{"*"}/\text{"\%"}, e1, e2) : \texttt{int}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O \rhd_{Expr} e : \texttt{int}}{O \rhd_{Expr} \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\ \frac{O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Expr} \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

$$[\textbf{Binary1}]\ \frac{O \rhd_{Expr} e1 : \texttt{int}, O \rhd_{Expr} e2 : \texttt{int}}{O \rhd_{Expr} \text{Binary}(\text{"+"}/\text{"-"}/\text{"*"}/\text{"\%"}, e1, e2) : \texttt{int}}$$

$$[\textbf{Binary2}]\ \frac{O \rhd_{Expr} e1 : \texttt{boolean}, O \rhd_{Expr} e2 : \texttt{boolean}}{O \rhd_{Expr} \text{Binary}(\text{"\&\&"}/\text{"||"}, e1, e2) : \texttt{boolean}}$$

Expression–Regel: Variablen

$$[\textbf{LocalOrFieldVar}]\ \frac{O \ \rhd_{Id} \ v : \theta}{O \ \rhd_{Expr} \ \text{LocalOrFieldVar}(\ v\ ) : \theta}$$

Expression–Regel: Variablen

$$[\textbf{LocalOrFieldVar}] \quad \frac{O \rhd_{Id} v : \theta}{O \rhd_{Expr} \text{LocalOrFieldVar}(v) : \theta}$$

$$[\textbf{InstVar}] \quad \frac{O \rhd_{Expr} re : \overline{\tau}, \qquad O_{\overline{\tau}} \rhd_{Id} v : \theta}{O \rhd_{Expr} \text{InstVar}(re, v) : \theta}$$

Beispiel InstVar

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

Beispiel InstVar

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$[\textbf{Ident}] \quad \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}$$

Beispiel InstVar

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$[\textbf{Ident}]\ \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}} \qquad [\textbf{LocalOrFieldVar}]\ \frac{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}{\{\, a : A \,\} \rhd_{Expr} \mathsf{LocalOrFieldVar}(\, a \,) : \mathtt{A}} \qquad \frac{}{O_{\mathtt{A}}}$$

Beispiel InstVar

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$\dfrac{\dfrac{\dfrac{}{\{\, a : A \,\}}\,[\textbf{Ident}]}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}\,[\textbf{LocalOrFieldVar}]}{\{\, a : A \,\} \rhd_{Expr} \mathsf{LocalOrFieldVar}(\, a \,) : \mathtt{A},} \qquad \dfrac{\dfrac{}{O_{\mathtt{A}}}\,[\textbf{Ident}]}{O_{\mathtt{A}} \rhd_{Id} \mathtt{attr} : \mathtt{int}}$$

Beispiel InstVar

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$\dfrac{\dfrac{\dfrac{}{\{\, a : A \,\}}\,[\textbf{Ident}]}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}\,[\textbf{LocalOrFieldVar}] \qquad \dfrac{}{O_{\mathtt{A}}}\,[\textbf{Ident}]}{\dfrac{\{\, a : A \,\} \rhd_{Expr} \mathsf{LocalOrFieldVar}(\, a \,) : \mathtt{A}, \qquad O_{\mathtt{A}} \rhd_{Id} \mathtt{attr} : \mathtt{int}}{\{\, a : A \,\} \rhd_{Expr} \textbf{InstVar}(\, \textbf{LocalOrFieldVar}(\, a \,), \mathit{attr} \,) : \mathtt{int}}\,[\textbf{InstVar}]}$$

Expression–Regel: Statement–Expressions

[**New**] $O \rhd_{Expr} \mathsf{New}(\theta) : \theta$

[1] $\leq^*$ ist die Subtypen–Relation

Expression–Regel: Statement–Expressions

$$[\textbf{New}] \qquad O \rhd_{Expr} \text{New}(\theta) : \theta$$

$$[\textbf{Assign}] \qquad \frac{O \rhd_{Expr} ve : \theta', O \rhd_{Expr} e : \theta}{O \rhd_{Expr} \text{Assign}(ve, e) : \theta'} \; \theta \leq^* \theta'^{1}$$

[1] $\leq^*$ ist die Subtypen–Relation

Expression–Regel: Statement–Expressions

$$[\textbf{New}] \qquad O \ \rhd_{Expr} \ \mathsf{New}(\,\theta\,) : \theta$$

$$[\textbf{Assign}] \qquad \frac{O \ \rhd_{Expr} \ ve : \theta', O \ \rhd_{Expr} \ e : \theta}{O \ \rhd_{Expr} \ \mathsf{Assign}(\,ve, e\,) : \theta'} \ \theta \leq^* \theta'^{1}$$

$$[\textbf{Method-Call}] \qquad \frac{\begin{array}{c} O \ \rhd_{Expr} \ re : \overline{\tau} \\ O_{\overline{\tau}} \ \rhd_{Id} \ m : \theta'_1 \times \ldots \times \theta'_n \rightarrow \theta \\ \forall 1 \leqslant i \leqslant n : \ O \ \rhd_{Expr} \ e_i : \theta_i \end{array}}{O \ \rhd_{Expr} \ \mathsf{MethodCall}(\,re, m, (e_1, \ldots, e_n)\,) : \theta} \ \theta_i \leq^* \theta'_i$$

[1] $\leq^*$ ist die Subtypen–Relation

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

[**Ident**] $$\frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}$$

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

$$[\textbf{Ident}]\ \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}} \qquad [\textbf{LocalOrFieldVar}]\ \frac{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}{\{\, a : A \,\} \rhd_{Expr} \mathrm{LocalOrFieldVar}(\, a \,) : \mathtt{A}} \qquad \frac{}{O_{\mathtt{A}}}$$

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\texttt{A}} = \{\, \texttt{attr} : \texttt{int},\ \texttt{meth} : \texttt{Boolean} \to \texttt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

$$[\textbf{Ident}]\ \frac{}{\{\, a : A \,\} \rhd_{Id} \texttt{a} : \texttt{A}} \qquad [\textbf{LocalOrFieldVar}]\ \frac{\{\, a : A \,\} \rhd_{Id} \texttt{a} : \texttt{A}}{\{\, a : A \,\} \rhd_{Expr} \text{LocalOrFieldVar}(\, a \,) : \texttt{A},} \qquad [\textbf{Ident}]\ \frac{O_{\texttt{A}}}{O_{\texttt{A}} \rhd_{Id} \texttt{meth} : \texttt{Boolean} \to \texttt{A}}$$

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\texttt{A}} = \{\, \texttt{attr} : \texttt{int},\ \texttt{meth} : \texttt{Boolean} \to \texttt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

$$[\textbf{Ident}]\ \frac{}{\{\, a : A \,\} \rhd_{Id} \texttt{a} : \texttt{A}}$$

$$[\textbf{LocalOrFieldVar}]\ \frac{\{\, a : A \,\} \rhd_{Id} \texttt{a} : \texttt{A}}{\{\, a : A \,\} \rhd_{Expr} \text{LocalOrFieldVar}(\, a \,) : \texttt{A},}$$

$$[\textbf{Ident}]\ \frac{O_{\texttt{A}}}{O_{\texttt{A}} \rhd_{Id} \texttt{meth} : \texttt{Boolean} \to \texttt{A}}$$

$$[\textbf{BoolLiteral}]\ \text{Bool}(\, \texttt{true} \,) : \texttt{boolean}$$

Beispiel MethodCall

```
class A {
    int attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{int},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

$$\dfrac{\dfrac{\dfrac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}\,[\textbf{Ident}]}{\{\, a : A \,\} \rhd_{Expr} \mathsf{LocalOrFieldVar}(\, a \,) : \mathtt{A},}\,[\textbf{LocalOrFieldVar}] \qquad \dfrac{\dfrac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id} \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A}}\,[\textbf{Ident}] \qquad [\textbf{BoolLiteral}]\ \mathsf{Bool}(\, \mathtt{true} \,) : \mathtt{boolean}}{\{\, a : A \,\} \rhd_{Expr} \mathbf{MethodCall}(\, \mathbf{LocalOrFieldVar}(\, a \,), \mathtt{meth}, \mathsf{Bool}(\mathtt{true}) \,) : \mathtt{A}}\,[\textbf{MethodCall}]$$

Statement–Regeln

$$[\textbf{Return}]\ \frac{O \rhd_{Expr} e : \theta}{O \rhd_{Stmt} \text{Return}(\, e \,) : \theta}$$

[2] upper bound

Statement–Regeln

$$[\textbf{Return}]\ \frac{O \rhd_{Expr} e : \theta}{O \rhd_{Stmt} \text{Return}(\, e \,) : \theta}$$

$$[\textbf{If}]\ \frac{O \rhd_{Stmt} s_1 : \theta_1, O \rhd_{Stmt} s_2 : \theta_2 \quad O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Stmt} \text{If}(\, e, s_1, s_2 \,) : \overline{\theta}, \textit{wobei}\ \overline{\theta} \in \textbf{UB}^2(\, \theta_1, \theta_2 \,)}$$

[2] upper bound

Statement–Regeln

$$[\textbf{Return}]\ \frac{O \rhd_{Expr} e : \theta}{O \rhd_{Stmt} \text{Return}(\, e \,) : \theta}$$

$$[\textbf{If}]\ \frac{O \rhd_{Stmt} s_1 : \theta_1, O \rhd_{Stmt} s_2 : \theta_2 \quad O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Stmt} \text{If}(\, e, s_1, s_2 \,) : \overline{\theta}, \textit{wobei}\ \overline{\theta} \in \textbf{UB}^2(\, \theta_1, \theta_2 \,)}$$

$$[\textbf{While}]\ \frac{O \rhd_{Expr} e : \texttt{boolean},\ O \rhd_{Stmt} \text{Block}(\, B \,) : \theta}{O \rhd_{Stmt} \text{While}(\, e, \text{Block}(\, B \,) \,) : \theta}$$

[2] upper bound

Statement–Regeln: Statement–Expressions

[**New**] $O \rhd_{Stmt} \mathsf{New}(\theta) : \texttt{void}$

Statement–Regeln: Statement–Expressions

$$[\textbf{New}] \qquad O \rhd_{Stmt} \text{New}(\theta) : \texttt{void}$$

$$[\textbf{Assign}] \qquad \frac{O \rhd_{Expr} ve : \theta', O \rhd_{Expr} e : \theta}{O \rhd_{Stmt} \text{Assign}(ve, e) : \texttt{void}} \; \theta \leq^* \theta'$$

Statement–Regeln: Statement–Expressions

$$[\textbf{New}] \qquad O \ \rhd_{Stmt} \ \mathsf{New}(\,\theta\,) : \texttt{void}$$

$$[\textbf{Assign}] \qquad \frac{O \ \rhd_{Expr} \ ve : \theta',\ O \ \rhd_{Expr} \ e : \theta}{O \ \rhd_{Stmt} \ \mathsf{Assign}(\,ve, e\,) : \texttt{void}} \ \theta \leq^* \theta'$$

$$[\textbf{Method-Call}] \qquad \frac{\begin{array}{c} O \ \rhd_{Expr} \ re : \overline{\tau} \\ O_{\overline{\tau}} \ \rhd_{Id} \ m : \theta'_1 \times \ldots \times \theta'_n \rightarrow \theta \\ \forall 1 \leqslant i \leqslant n : \ O \ \rhd_{Expr} \ e_i : \theta_i \end{array}}{O \ \rhd_{Stmt} \ \mathsf{MethodCall}(\,re, m, (e_1, \ldots, e_n)\,) : \texttt{void}} \ \theta_i \leq^* \theta'_i$$

Block–Statement Regeln

$$[\textbf{BlockInit}] \quad \frac{O \ \rhd_{Stmt} \ stmt : \theta}{O \ \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

Block–Statement Regeln

$$[\textbf{BlockInit}]\quad \frac{O \rhd_{Stmt} \ stmt : \theta}{O \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

$$[\textbf{Block}]\quad \frac{O \rhd_{Stmt} \ s_1 : \theta,\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta},\ \textit{wobei}\ \overline{\theta} \in \textbf{UB}(\ \theta, \theta'\ )}$$

Block–Statement Regeln

$$[\textbf{BlockInit}]\ \frac{O \rhd_{Stmt} \ stmt : \theta}{O \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

$$[\textbf{Block}]\ \frac{O \rhd_{Stmt} \ s_1 : \theta,\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta},\ wobei\ \overline{\theta} \in \textbf{UB}(\ \theta, \theta'\ )}$$

$$[\textbf{Blockvoid}]\ \frac{\begin{array}{c} O \rhd_{Stmt} \ s_1 : \texttt{void}, \\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta \end{array}}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \theta}$$

Block–Statement Regeln

$$[\textbf{BlockInit}]\ \frac{O \rhd_{Stmt} \ stmt : \theta}{O \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

$$[\textbf{Block}]\ \frac{O \rhd_{Stmt} \ s_1 : \theta,\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta},\ \textit{wobei}\ \overline{\theta} \in \textbf{UB}(\ \theta, \theta'\ )}$$

$$[\textbf{Blockvoid}]\ \frac{\begin{array}{c} O \rhd_{Stmt} \ s_1 : \texttt{void}, \\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta \end{array}}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \theta}$$

$$[\textbf{Block-Local-VarDecl}]\ \frac{O \backslash \{\ v : \theta'\ \} \cup \{\ v : \overline{\theta}\ \} \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta}{O \rhd_{Stmt} \ \text{Block}(\ \text{LocalVarDecl}(\ v, \overline{\theta}\ ); s_2; \ldots; s_n;\ ) : \theta}$$

Beispiel I

```
while (true) {
    return 1;
}
```

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$[\textbf{Int-Literal}] \qquad O \rhd_{Expr} \text{Integer}(1) : \texttt{int}$$

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\frac{[\textbf{Int-Literal}]}{[\textbf{Return}]}\ \frac{O \rhd_{Expr} \text{Integer}(1) : \texttt{int}}{O \rhd_{Stmt} \text{Return}(\text{Integer}(1)) : \texttt{int}}$$

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\dfrac{\dfrac{[\textbf{Int-Literal}]\quad O \rhd_{Expr} \text{Integer}(1) : \texttt{int}}{[\textbf{Return}]\quad O \rhd_{Stmt} \text{Return}(\text{Integer}(1)) : \texttt{int}}}{[\textbf{Block-Init}]\quad O \rhd_{Stmt} \text{Block}(\text{Return}(\text{Integer}(1))) : \texttt{int}}$$

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$[\textbf{Bool-Literal}]\ O \rhd_{Expr} \text{Bool}(\ \mathit{True}\ ) : \texttt{boolean}$$

$$\dfrac{[\textbf{Int-Literal}]\ O \rhd_{Expr} \text{Integer}(\ 1\ ) : \texttt{int}}{[\textbf{Return}]\ O \rhd_{Stmt} \text{Return}(\ \text{Integer}(\ 1\ )\ ) : \texttt{int}}$$

$$\dfrac{[\textbf{Block-Init}]\ O \rhd_{Stmt} \text{Return}(\ \text{Integer}(\ 1\ )\ )\ ) : \texttt{int}}{O \rhd_{Stmt} \text{Block}(\ \text{Return}(\ \text{Integer}(\ 1\ )\ )\ ) : \texttt{int}}$$

Beispiel I

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\dfrac{[\textbf{Bool-Literal}]\ O \rhd_{Expr} \text{Bool}(\ \mathit{True}\ ) : \texttt{boolean} \qquad \dfrac{[\textbf{Int-Literal}]\ O \rhd_{Expr} \text{Integer}(\ 1\ ) : \texttt{int}}{[\textbf{Return}]\ O \rhd_{Stmt} \text{Return}(\ \text{Integer}(\ 1\ )\ ) : \texttt{int}} \quad [\textbf{Block - Init}]}{[\textbf{while}]\ O \rhd_{Stmt} \text{While}(\ \text{Bool}(\ \mathit{True}\ ), \text{Block}(\ \text{Return}(\ \text{Integer}(\ 1\ )\ )\ )\ ) : \texttt{int}}$$

Beispiel II

```
class A { }
```

```
class B { }
```

```
{ return new A();
  new A();
  return new B(); }
```

[3] analog zu $O \rhd_{Expr}$ **Return**(New(B)) : B hergeleitet

Beispiel II

```
class A { }
```

```
class B { }
```

```
{ return new A();
  new A();
  return new B(); }
```

$O = \emptyset$

[3] analog zu $O \rhd_{Expr}$ **Return**(New(B)) : B hergeleitet

Beispiel II

```
class A { }

{ return new A();
  new A();
  return new B(); }
```

```
class B { }
```

$O = \emptyset$

[**New**] $O \rhd_{Expr} \text{New}(\, B \,) : \texttt{B}$

[3] analog zu $O \rhd_{Expr}$ **Return**(New(B)) : B hergeleitet

Beispiel II

```
class A { }

{ return new A();
  new A();
  return new B(); }
```

```
class B { }
```

$O = \emptyset$

$$\frac{O \rhd_{Expr} \mathsf{New}(B) : \mathtt{B}}{O \rhd_{Stmt} \mathsf{Return}(\mathsf{New}(B)) : \mathtt{B}} \begin{matrix}[\textbf{New}] \\ [\textbf{Return}]\end{matrix}$$

[3] analog zu $O \rhd_{Expr}$ **Return**(New(B)) : B hergeleitet

Beispiel II

```
class A { }

{ return new A();
  new A();
  return new B(); }
```

```
class B { }
```

$O = \emptyset$

$$\dfrac{\dfrac{\dfrac{O \rhd_{Expr} \text{New}(B) : \texttt{B}}{O \rhd_{Stmt} \text{Return}(\text{New}(B)) : \texttt{B}}\ [\textbf{Return}]}{O \rhd_{Stmt} \text{Block}(\text{Return}(\text{New}(B))) : \texttt{B}}\ [\textbf{Block-Init}]}\ [\textbf{New}]$$

[3] analog zu $O \rhd_{Expr} \textbf{Return}(\text{New}(B)) : \texttt{B}$ hergeleitet

Beispiel II

```
class A { }
```

```
class B { }
```

```
{ return new A();
  new A();
  return new B(); }
```

$O = \emptyset$

[**New**] $O \rhd_{Stmt} \text{New}(A) : \texttt{void}$

$$\dfrac{\dfrac{\dfrac{}{O \rhd_{Expr} \text{New}(B) : \texttt{B}}\,[\textbf{New}]}{O \rhd_{Stmt} \text{Return}(\text{New}(B)) : \texttt{B}}\,[\textbf{Return}]}{O \rhd_{Stmt} \text{Block}(\text{Return}(\text{New}(B))) : \texttt{B}}\,[\textbf{Block-Init}]$$

[3] analog zu $O \rhd_{Expr} \textbf{Return}(\text{New}(B)) : \texttt{B}$ hergeleitet

Beispiel II

```
class A { }

{ return new A();
  new A();
  return new B(); }
```

```
class B { }
```

$O = \emptyset$

$$\dfrac{[\textbf{Block-void}]\ \dfrac{[\textbf{New}]\ O \rhd_{Stmt} \text{New}(A) : \texttt{void} \qquad [\textbf{Block-Init}]\ \dfrac{[\textbf{Return}]\ \dfrac{[\textbf{New}]\ O \rhd_{Expr} \text{New}(B) : \texttt{B}}{O \rhd_{Stmt} \text{Return}(\text{New}(B)) : \texttt{B}}}{O \rhd_{Stmt} \text{Block}(\text{Return}(\text{New}(B))) : \texttt{B}}}{O \rhd_{Stmt} \text{Block}(\text{New}(A); \text{Return}(\text{New}(B))) : \texttt{B}}$$

[3] analog zu $O \rhd_{Expr} \textbf{Return}(\text{New}(B)) : \texttt{B}$ hergeleitet

Beispiel II

```
class A { }

{ return new A();
  new A();
  return new B(); }
```

```
class B { }
```

$O = \emptyset$

$$\dfrac{\dfrac{\dfrac{}{O \rhd_{Stmt} \texttt{New}(A) : \texttt{void}}\,[\textbf{New}] \quad \dfrac{\dfrac{\dfrac{}{O \rhd_{Expr} \texttt{New}(B) : \texttt{B}}\,[\textbf{New}]}{O \rhd_{Stmt} \texttt{Return}(\texttt{New}(B)) : \texttt{B}}\,[\textbf{Return}]}{O \rhd_{Stmt} \texttt{Block}(\texttt{Return}(\texttt{New}(B))) : \texttt{B}}\,[\textbf{Block-Init}]}{O \rhd_{Stmt} \texttt{Return}(\texttt{New}(A)) : \texttt{A}^3 \quad O \rhd_{Stmt} \texttt{Block}(\texttt{New}(A); \texttt{Return}(\texttt{New}(B))) : \texttt{B}}\,[\textbf{Block-void}]}{O \rhd_{Stmt} \texttt{Block}(\texttt{Return}(\texttt{New}(A)); \texttt{New}(A); \texttt{Return}(\texttt{New}(B))) : \texttt{Object}}\,[\textbf{Block}]$$

wobei $\textbf{UB}(\texttt{A}, \texttt{B}) = \texttt{Object}$

[3] analog zu $O \rhd_{Expr} \textbf{Return}(\texttt{New}(B)) : \texttt{B}$ hergeleitet

Datenstruktur typisierter Expressions

```
type Type = String

-- data Type = TVar(String)              -- Typvariable
--           | TC(String, [Type])        -- Typkonstruktor
--           | WC                        -- Wildscard
--           | WC_Super(Type)            -- Super-Wildscard
--           | WC_Extends(Type)          -- Extends-Wildcard
```

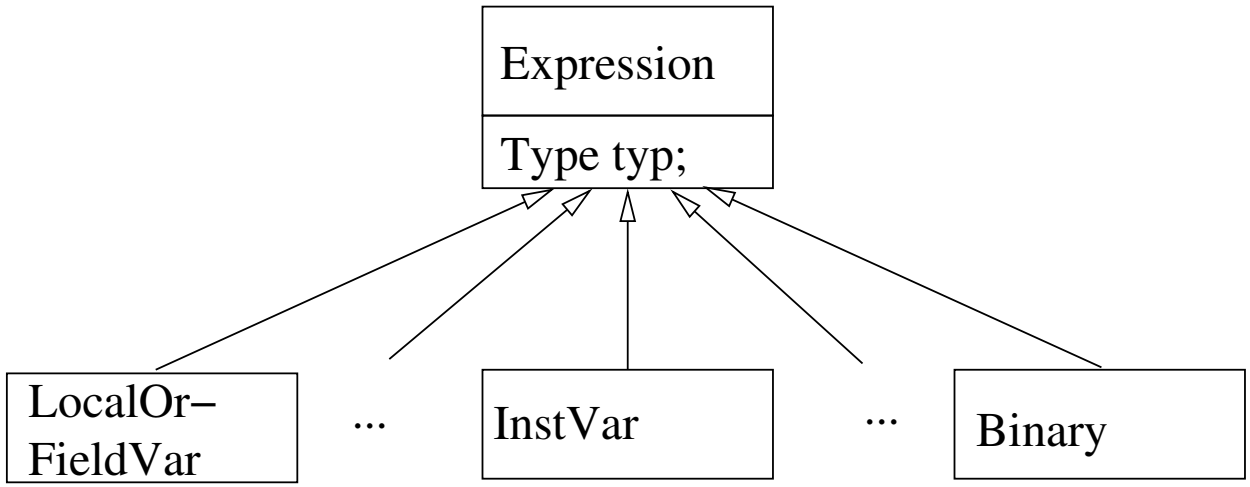
Datenstruktur typisierter Expressions

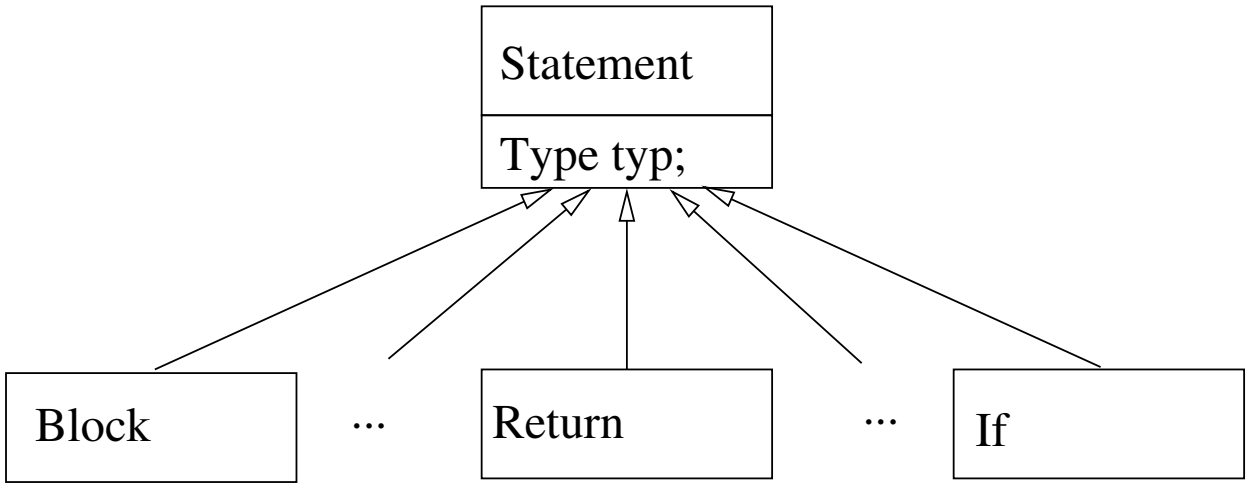
```
type Type = String

-- data Type = TVar(String)              -- Typvariable
--           | TC(String, [Type])        -- Typkonstruktor
--           | WC                        -- Wildscard
--           | WC_Super(Type)            -- Super-Wildscard
--           | WC_Extends(Type)          -- Extends-Wildcard
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)
          | TypedExpr(Expr, Type)
```

Datenstruktur typisierter Statements

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

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

Datenstruktur typisierter Expressions/Statements





Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung:

```
Return(
  TypedExpr(
        Binary("+",
                TypedExpr(Integer(1), "int"),
                TypedExpr(Integer(2), "int")),
         "int"))
```

Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

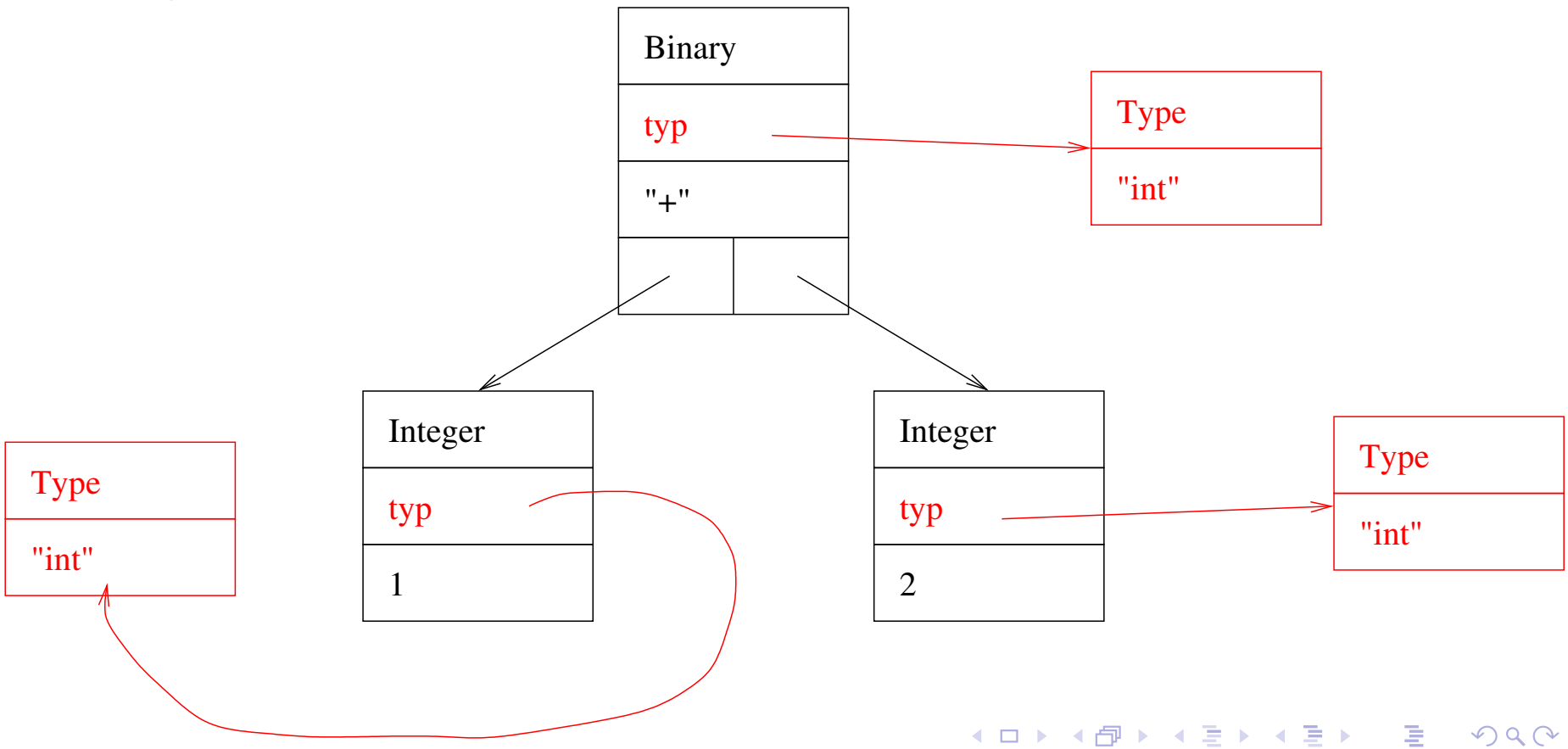
Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung:



LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) { return v + x; }
}
```

LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) { return v + x; }
}
```

Erg. der Parsers:

```
Return(Binary("+",
                  LocalOrFieldVar("v")      ← FieldVar
                  LocalOrFieldVar("x")))    ← LocalVar
```

LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) { return v + x; }
}
```

Erg. der Parsers:

```
Return(Binary("+",
                  LocalOrFieldVar("v")      ← FieldVar
                  LocalOrFieldVar("x")))    ← LocalVar
```

Typisierung:

```
Return(
  TypedExpr(
         Binary("+",
                   TypedExpr(LocalOrFieldVar("v"), "int"),
                   TypedExpr(LocalOrFieldVar("x"), "int")),
          "int"))
```

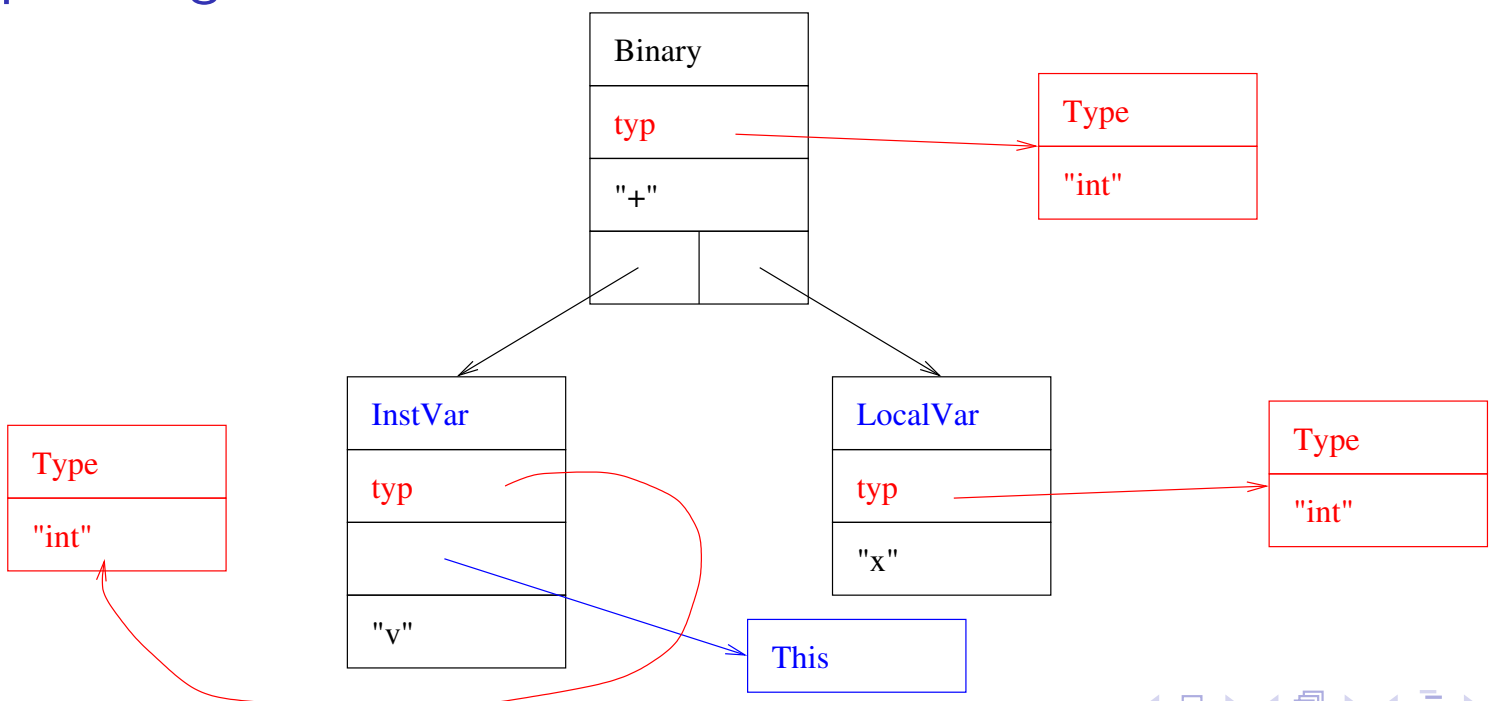
LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) { return v + x; }
}
```

Erg. der Parsers:

```
Return(Binary("+",
                  LocalOrFieldVar("v")      ← FieldVar
                  LocalOrFieldVar("x")))    ← LocalVar
```

Typisierung:



InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(Cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(Cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",
                  [InstVar(LocalOrFieldVar("x"), "v"),
                   LocalOrFieldVar("b")]))
```

InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(Cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",
                  [InstVar(LocalOrFieldVar("x"), "v"),
                   LocalOrFieldVar("b")]))
```

Typisierung:

```
Return(TypedExpr(
  MethodCall(TypedExpr(LocalOrFieldVar("x"), "Cl2"), "m2",
             [TypedExpr(
                InstVar(TypedExpr(LocalOrFieldVar("x"), "Cl2"),
                        "v"), "Cl3"),
              TypedExpr(LocalOrFieldVar("b"), "int")]), "char"))
```

Semantische Analyse/Algorithmus `typecheck` (Haskell)

```
typecheckExpr:: Expr -> [(String, Type)] -> [Class] -> Expr
typecheckStmt:: Stmt -> [(String, Type)] -> [Class] -> Stmt
```

1. Argument: Ungetypter Ausdruck/Statement
2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
3. Argument: Alle sichtbaren Klassen mit Fields und Methoden

Ergebnis: getypeter Ausdruck/Statement

Semantische Analyse/Algorithmus `typecheck` (Haskell)

```
typecheckExpr:: Expr -> [(String, Type)] -> [Class] -> Expr
typecheckStmt:: Stmt -> [(String, Type)] -> [Class] -> Stmt
```

1. Argument: Ungetypter Ausdruck/Statement
2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
3. Argument: Alle sichtbaren Klassen mit Fields und Methoden

Ergebnis: getypeter Ausdruck/Statement

Ablauf: Lauf über alle Methoden aller Klassen:

- Check ob Variablen/Methoden deklariert
- Check, ob die Typen der Metdodenaufrufe/Zuweisung korrekt sind
- Einfügen der Typisierungen

Beispiel: typeCheck für IF

$$[\textbf{If}]\ \frac{O \rhd_{Stmt} s_1 : \theta_1,\ O \rhd_{Stmt} s_2 : \theta_2 \quad O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Stmt} \textsf{If}(e, s_1, s_2) : \overline{\theta},\ \textit{wobei}\ \overline{\theta} \in \textbf{UB}(\theta_1, \theta_2)}$$

```
typecheckStmt :: Stmt -> [(String, Type)] -> [Class] -> Stmt
typecheckStmt (If(be, ifs, Nothing)) symtab cls =
  let
    bexp = typecheckExpr be symtab cls
    ifstmt = typecheckStmt ifs symtab cls
  in
    if ((getTypeFromExpr bexp) == "boolean") then
      TypedStmt(If(bexp, ifstmt, Nothing), getTypeFromStmt ifstmt)
    else
     error "boolean expected"

getTypeFromExpr :: Expr -> Type
getTypeFromExpr (TypedExpr(_, typ)) = typ

getTypeFromStmt :: Stmt -> Type
getTypeFromStmt (TypedStmt(_, typ)) = typ
```

Semantische Analyse/Algorithmus typecheck (Java)

```
Type typeCheck(Map<String, Type> localVars, Vector<Class>
classes)
```

1. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
2. Argument: Klassen mit Attributen (Fields) und Methoden

Ergebnis: Typ des Ausdrucks/Statements

Semantische Analyse/Algorithmus typecheck (Java)

```
Type typeCheck(Map<String, Type> localVars, Vector<Class>
classes)
```

1. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
2. Argument: Klassen mit Attributen (Fields) und Methoden

Ergebnis: Typ des Ausdrucks/Statements

Ablauf: Lauf über alle Methoden, alle Statements und alle Expressions der Klasse:

- Check ob Variablen/Methoden deklariert
- Check, ob die Typen der Metdodenaufrufe/Zuweisung korrekt sind
- Einfügen der Typisierungen
- Ersetzen von `LocalOrFieldVar` zu `InstVar` bzw. `LocalVar`

Beispiel: typeCheck für IF

$$[\textbf{If}]\ \frac{O \rhd_{Stmt} s_1 : \theta_1,\ O \rhd_{Stmt} s_2 : \theta_2 \quad O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Stmt} \textsf{If}(e, s_1, s_2) : \overline{\theta},\ \textit{wobei}\ \overline{\theta} \in \textbf{UB}(\theta_1, \theta_2)}$$

```
class Statement {
 Type typ;
 abstract Type typeCheck(Map<String, Type> localVars, Vector<Class>
      classes);
}

class If extends Statement {
  Expression cond;
  Statement ifStmt;
  Statement elseStmt;

  Type typeCheck(Map<String, Type> localVars, Vector<Class> classes) {
    if (cond.typeCheck(localVars, classes).equals(new Type("boolean"))
        && ifStmt.typeCheck(localVars, classes)
                 .equals(elseStmt.typeCheck(localVars, classes))) {
        typ = ifStmt.typeCheck(localVars, classes);
        return typ;  }
    else {          //error                    }
  }
```