

# Compilerbau





Martin Plümicke  
Andreas Stadelmeier

SS 2025





# 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 Kenntnissen der Grundvorlesung vermittelt. Im 2. Teil der Lehrveranstaltungen werden die Studierenden in Gruppenarbeit einen Mini-Java-Compiler mit den gelernten Techniken implementieren.

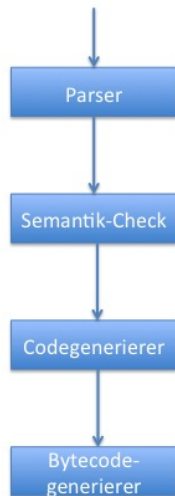
# Literatur

-  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).

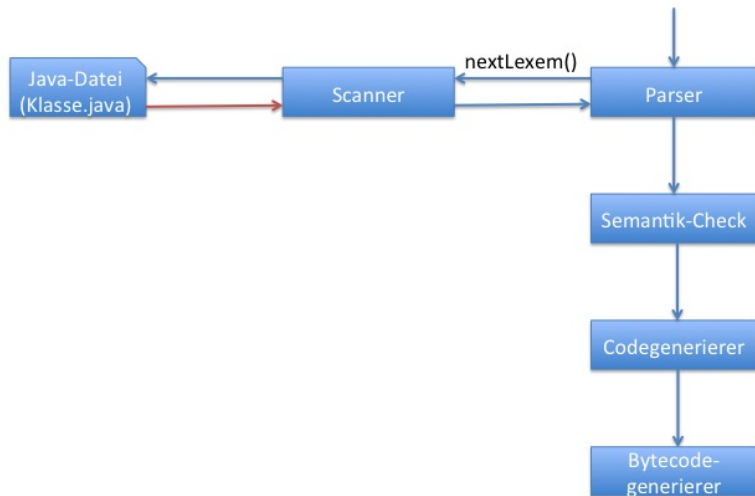
# Literatur II

-  James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.  
*The Java<sup>®</sup> Language Specification.*  
The Java series. Addison-Wesley, Java SE 8 edition, 2014.
-  Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.  
*The Java<sup>®</sup> 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.

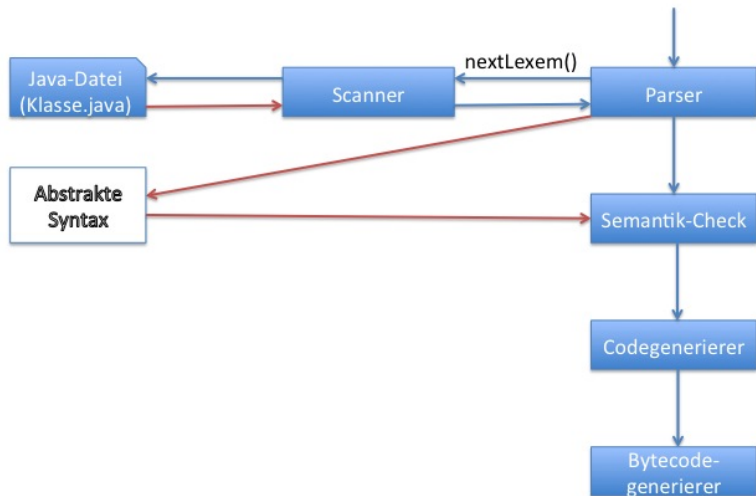
# Compiler Überblick



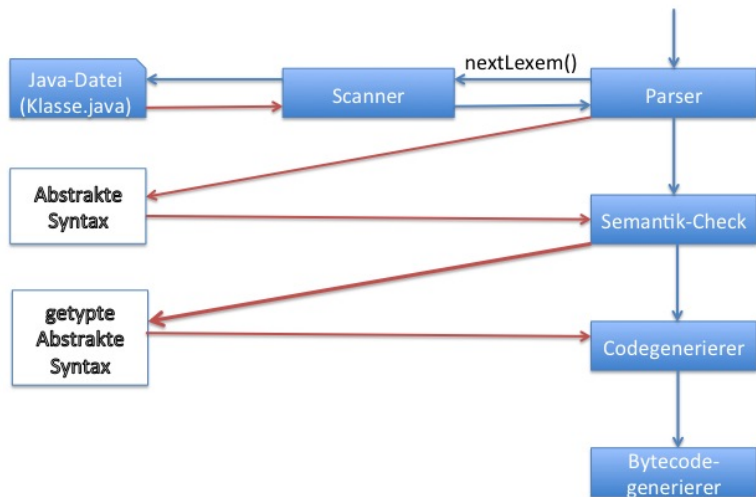
# Compiler Überblick



# Compiler Überblick

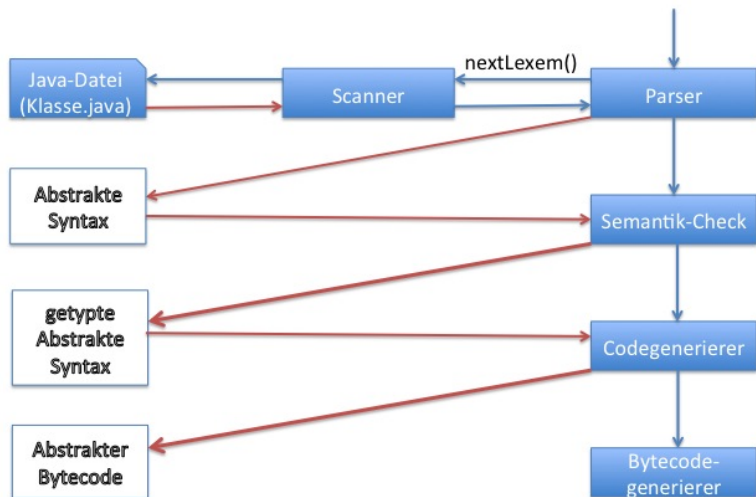


# Compiler Überblick

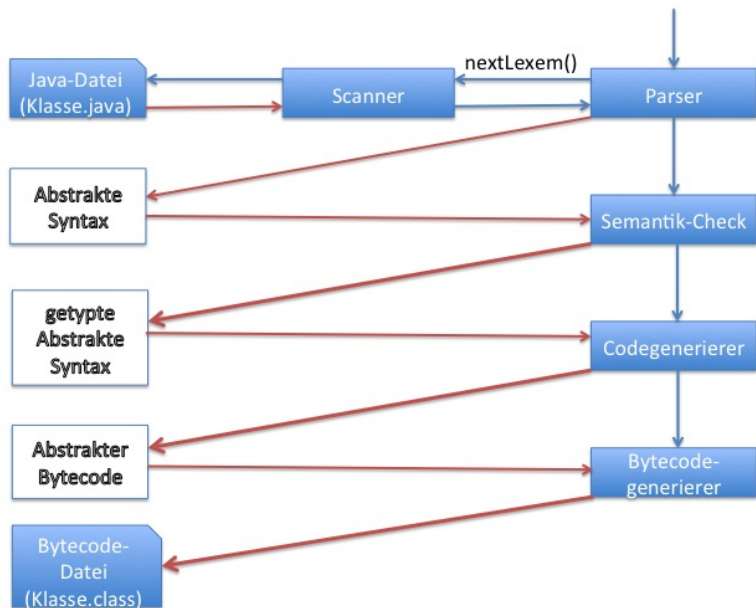




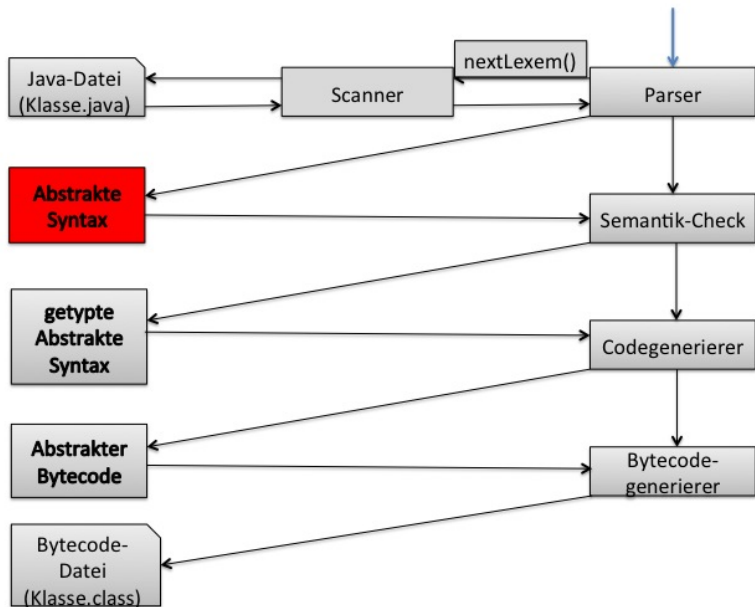
# Compiler Überblick



# 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 *Mini funktionale Expressions*

```
data MiniFunkExpr =  
    Let String MiniFunkExpr MiniFunkExpr  
  | Plus MiniFunkExpr MiniFunkExpr  
  | Const Int  
  | Var String  
  deriving (Eq, Show)
```

# Transformation in abstrakte Syntax

```
expr :: Parser Token MiniFunkExpr
expr = (texp ++ expr')
```

```
expr' :: Parser Token (Maybe MiniFunkExpr)
expr' =
  (((satisfy (isSym '+')) ++ texp ++ expr')
```

```
||| succeed Nothing
```

```
texp :: Parser Token MiniFunkExpr
texp = (((lexem LetToken) ++ (satisfy isVar) ++
  (satisfy (isSym '=')) ++ expr ++ (lexem InToken) ++ expr)
```

```
||| ((satisfy isVar)
```

```
||| ((satisfy isInt)
```

# Transformation in abstrakte Syntax

```
expr :: Parser Token MiniFunkExpr
expr = (texp ++ expr')
      <<< \(e1, e2) ->
          if (e2 == Nothing) then e1
            else Plus e1 (fromJust e2)

expr' :: Parser Token (Maybe MiniFunkExpr)
expr' =
  (((satisfy (isSym '+')) ++ texp ++ expr')
   <<< (\(_, (e1, e2)) ->
       if (e2 == Nothing) then Just e1
         else Just (Plus e1 (fromJust e2))))
  ||| succeed Nothing

texp :: Parser Token MiniFunkExpr
texp = (((lexem LetToken) ++ (satisfy isVar) ++
  (satisfy (isSym '=') ++ expr ++ (lexem InToken) ++ expr)
  <<< (\(_, (VarToken id, (_, (e, (_, e2)))))) -> (Let id e e2)))
  ||| ((satisfy isVar) <<< (\(VarToken id) -> Var id))
  ||| ((satisfy isInt) <<< (\(IntToken n) -> Const n))
```

# Grammatik und Übersetzung

## Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =  
    Let String MiniFunkExpr MiniFunkExpr  
  | Plus MiniFunkExpr MiniFunkExpr  
  | Const Int  
  | Var String  
deriving (Eq, Show)
```



# Grammatik und Übersetzung

## Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =  
    Let String MiniFunkExpr MiniFunkExpr  
  | Plus MiniFunkExpr MiniFunkExpr  
  | Const Int  
  | Var String  
  deriving (Eq, Show)
```

## Happy-File mit

```
%% -- aus yacc-Tradition
```

```
expr          : Let Var Assign expr In expr { Let $2 $4 $6 }  
              | expr Plus expr { Plus $1 $3 }  
              | Var { Var $1 }  
              | Int { Const $1 }
```

# Grammatik und Übersetzung

## Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =  
    Let String MiniFunkExpr MiniFunkExpr  
  | Plus MiniFunkExpr MiniFunkExpr  
  | Const Int  
  | Var String  
  deriving (Eq, Show)
```

## Happy-File mit

```
%% -- aus yacc-Tradition
```

```
expr          : Let Var Assign expr In expr { Let $2 $4 $6 }  
              | expr Plus expr { Plus $1 $3 }  
              | Var { Var $1 }  
              | Int { Const $1 }
```

- ▶ 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

```
Let Var Assign expr In expr { Let $2 $4 $6 }
```

bedeutet: Beim *reduce* wird ein **Let**-Element erzeugt, das als Argumente

1. das Argument des Terminals Var (\$2) und
2. das Ergebnis von `expr` (\$4) und
3. das Ergebnis von `expr` (\$6) und

hat.

## Beispiel

```
Let Var Assign expr In expr { Let $2 $4 $6 }
```

bedeutet: Beim *reduce* wird ein **Let**-Element erzeugt, das als Argumente

1. das Argument des Terminals Var (\$2) und
2. das Ergebnis von expr (\$4) und
3. das Ergebnis von expr (\$6) und

hat.

```
let x = 2 in x
```

gibt das Paar **Let** "x" (Const 2) (Var "x") zurück.

# 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:

StmtExpr Stmt (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]
```