Objektorientierter Übersetzerbau

Dipl.-Inform. Peter Knauber

Universität Kaiserslautern

FB Informatik, AG Programmiersprachen und Compiler

Inhaltsübersicht

- Vorbemerkungen zu Oberon-2, Java, W-Lisp
- Traditioneller Übersetzerbau ⇒ objektorientierter Übersetzerbau
- Eine Bibliothek mit Sprachkonstrukten
- Das OCC-System:
  - Phasen der Übersetzung
  - Beispiele, Erfahrungen, Probleme

Oberon-2: Klassen

- Oberon-2 enthält Vererbung: erweiterbare Records

```
TYPE R0     =  RECORD
       f1   :  INTEGER;
       f2   :  CHAR;
     END;
     R1     =  RECORD ( R0 )
       f2   :  REAL;        (*nur in untersch. Modulen erlaubt*)
       f3   :  BOOLEAN;
     END;
```

- Der Record „R1“ erbt das Feld „f1“ von seinem Basistyp „R0“.
  Die Überdeckung von Feld „f2“ ist nur erlaubt, falls beide Records in unterschiedlichen Modulen definiert sind und „f2“ von „R0“ nicht exportiert wird.
  Zusätzlich enthält der Record „R1“ das Feld „f3“.

Oberon-2: Methoden

- Oberon-2 enthält Methoden: typgebundene Prozeduren
  Diese werden wie Recordfelder behandelt.

```
PROCEDURE ( VAR r : R0 ) p1 ( … );      BEGIN … END p1;
PROCEDURE ( VAR r : R0 ) p2 ( … );      BEGIN … END p2;
PROCEDURE ( VAR r : R1 ) p2 ( … );      BEGIN … END p2;
PROCEDURE ( VAR r : R1 ) p3 ( … );      BEGIN … END p3;
```

- Die reguläre Parameterliste (hier: „…“) muß übereinstimmen.

- Der ausgezeichnete erste Parameter wird als „Receiver“ bezeichnet.
  Er muß ein Referenzparameter von einem Recordtyp oder ein Zeiger (auf einen Recordtyp) sein.

Oberon-2: Methoden-Aktivierung

- Da der Receiver immer durch einen Zeiger realisiert wird, können Erweiterungstypen (Unterklassen) übergeben werden.

  Man unterscheidet den „statischen“ und den „dynamischen“ Typ eines solchen Parameters.

  Es wird statisch überprüft, daß der Typ des aktuellen Parameters zuweisbar ist:

  VAR o0 : R0; o1 : R1;

	(* Stat. Typ	Dyn. Typ *)
o0.p1(…)	R0	R0
o0.p2(…)	R0	R0
o0.p3(…)	verboten!	
o1.p1(…)	R0	R1
o1.p2(…)	R1	R1
o1.p3(…)	R1	R1

Merkmale von Java

- Einfachvererbung

- Aufruf der Methode einer Oberklasse (Supercall): super.*methodenname*(…);

- Abstrakte Klassen: sind nicht instanziierbar

  abstract class C { … }

- Abstrakte Methoden

  abstract *methodenname* (…);

  bewirken, daß die umgebende Klasse abstrakt ist.

  Sie müssen in einer Unterklasse implementiert werden, damit diese instanziiert werden kann.

W-Lisp / CLOS

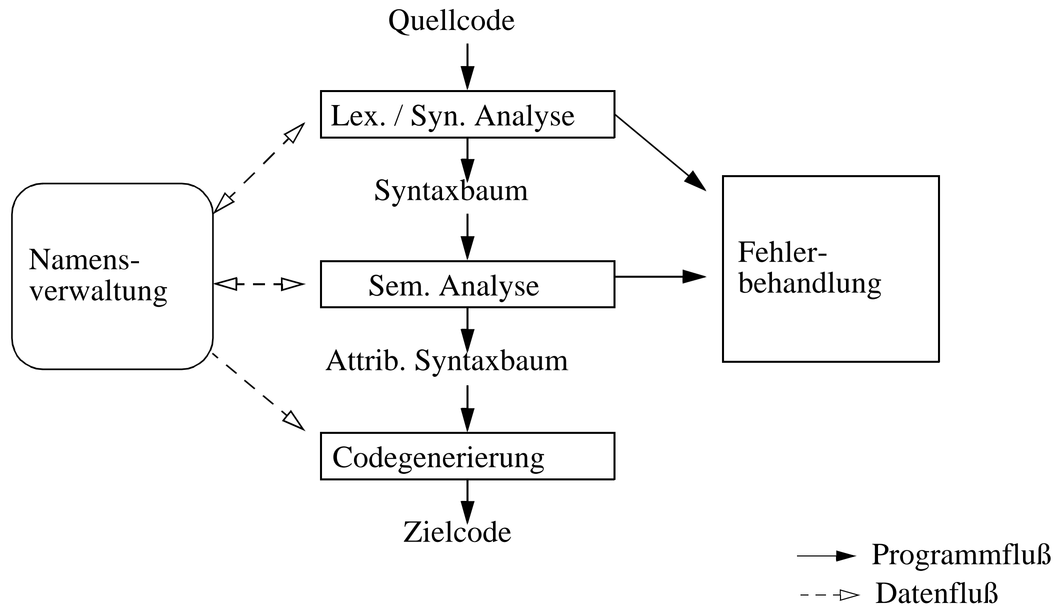
- Implementierungssprache für den OCC-Prototyp: W-Lisp
  W-Lisp ist ein Präprozessor für Common Lisp / CLOS
  und bietet besser lesbare Konstrukte.
- Möglichkeiten der Sprache:
  - Listen (aus Common Lisp)
  - Abstrakte Klassen
  - Abstrakte Methoden
  - Mehrfachvererbung

W-Lisp - Beispiel

```
ABSTRACT-CLASS X IS
    a   =   3
END-CLASS
CLASS Y INHERITS X IS
    b
END-CLASS
METHOD m ( o OF X ) IS
    IF o.a > 4 THEN
        …
    FI
    <o.b v1 ..v2> <- {1 2 3 4}
END-METHOD
```

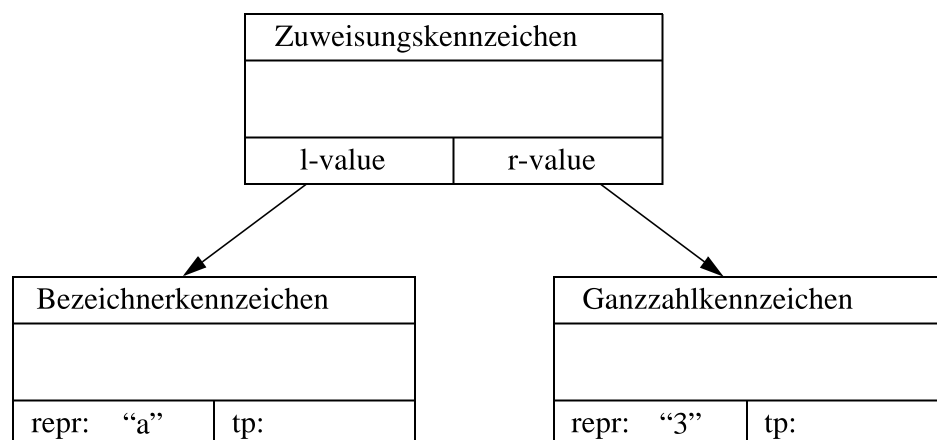
```
(defclass x ()
    (   (a :initial-content 3))
)
(defclass y ( x )
    (   b ))
)
(defmethod m ( (o x) )
    (let ( v1 v2 tmp )
        (cond       (   (> (a o) 4)
                         … ))
        (setf tmp (list 1 2 3 4)
              (b o) (first tmp)
              v1 (second tmp)
              v2 (rest tmp))))
```

Traditionelle phasenorientierte Übersetzer



Beispiel aus Pascal

(Abstrakter) Syntaxbaum für „a := 3“



Namenstabelle	
“a”	VAR intTp

Darstellung des Syntaxbaums in Pascal: Typen

```
TYPE NodePtr             =   ^ Node;
     NodeTp              =   RECORD
       CASE tag          :   INTEGER OF
         …
         idTag           :   (   tp          :   PascalTp;
                                 repr        :   STRING;
         intTag          :   (   tp          :   PascalTp;
                                 repr        :   STRING;
                             )
         assignTag       :   (   lValue,
                                 rValue      : NodePtr; )
         …
     END;
```

Darstellung des Syntaxbaums in Pascal: Anweisungen

```
PROCEDURE analyze ( n : NodeTp );
VAR entry : NameTableEntryPtr;
BEGIN
   CASE n.tag OF
      idTag          : BEGIN
         entry := search( n.repr, … );
         n.tp  := entry^.tp; END;
      intTag         : n.tp  := intTp;
      assignTag      : BEGIN
         analyze( n.lValue^ );
         analyze( n.rValue^ );
         IF NOT compatible( n.lValue^.tp, n.rValue^.tp ) THEN … END;
   END;
END; (*analyze*)
```

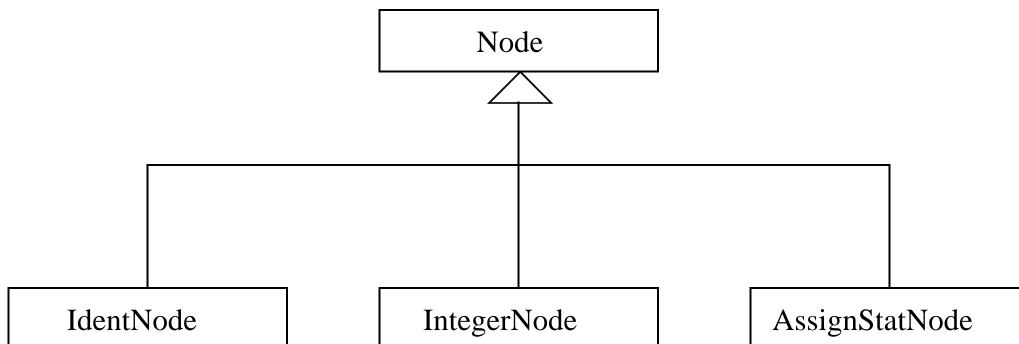

Nachteile dieser Implementierung

- Im Falle zusätzlicher Knotentypen

- Im Falle von Modifikationen der Semantik:
  zusätzliche Felder und deren Behandlung müssen berücksichtigt werden

- Allgemein: unübersichtliche Programme
  wegen langer Typ- und Prozedur-Vereinbarungen

Lösung: Objektorientierte Darstellung



Notation: UML

Darstellung in Oberon-2

```
TYPE  NodePtr          =  POINTER TO Node;
      Node             =  RECORD
      END;
      IdentNode        =  RECORD ( Node )
         repr          :  ARRAY OF CHAR;
         tp            :  INTEGER;
      END;
      IntNode          =  RECORD ( Node )
         repr          :  ARRAY OF CHAR;
         tp            :  INTEGER;
      END;
      AssignNode       =  RECORD ( Node )
         lValue, rValue :  NodePtr;
      END;
```

Zugehörige Anweisungen (1)

```
PROCEDURE ( VAR n : Node ) analyze;
BEGIN
    (* solch ein Knoten darf nicht auftreten → Fehlermeldung! *)
END analyze;

PROCEDURE ( VAR n : IdentNode ) analyze;
VAR entry : NameTableEntryPtr;
BEGIN
    entry := search( n.repr );
    n.tp := entry.tp;
END analyze;
```


Zugehörige Anweisungen (2)

```
PROCEDURE ( VAR n : IntNode ) analyze;
BEGIN
    n.tp := intTp;
END analyze;

PROCEDURE ( VAR n : AssignNode ) analyze;
BEGIN
    n.lValue.analyze();
    n.rValue.analyze();
    IF NOT compatible( n.lValue.tp(), n.rValue.tp() ) THEN
        …
    END;
END analyze;
```

Problem: Zugriff auf den Typ von Ausdrücken

o Oberon-2 macht eine statische Überprüfung, ob der Zugriff auf ein Attribut / eine Methode möglich ist. Beispiel:

```
TYPE  AssignNode          =   RECORD ( Node )
          lValue, rValue  :   NodePtr; END;

PROCEDURE ( VAR n : AssignNode ) analyze; …
    IF NOT compatible( n.lValue.tp, n.rValue.tp ) THEN …
```

o Das Feld „tp“ existiert nicht im Knoten „Node“, daher

```
error    83:    undefined record field
line     53:    n.tp  := entry.tp;
                  ^
```

‰ Lösung: im allgemeineren Knoten vorsehen

Typbestimmung von Ausdrücken: Alternative 1

Realisierung durch ein Attribut:

```
TYPE    Node    =   RECORD
          tp    :   INTEGER;
        END;
```

o Zugriff:

```
PROCEDURE ( VAR n : AssignNode ) analyze;
    …
    IF NOT compatible( n.lValue.tp, n.rValue.tp ) THEN …
```

Â Nachteil:

Das Attribut ist nur für Ausdrücke (hier: Bezeichner, ganze Zahlen) sinnvoll belegt, steht aber in allen Knoten zur Verfügung.

Typbestimmung von Ausdrücken: Alternative 2

Realisierung durch eine Methode:

```
PROCEDURE ( VAR n : Node ) tp () : INTEGER;
BEGIN
    RETURN -1; (* Fehlerbehandlung *)
END tp;

PROCEDURE ( VAR n : IdentNode ) tp () : INTEGER;
BEGIN
    RETURN n.localTp;
END tp;
```



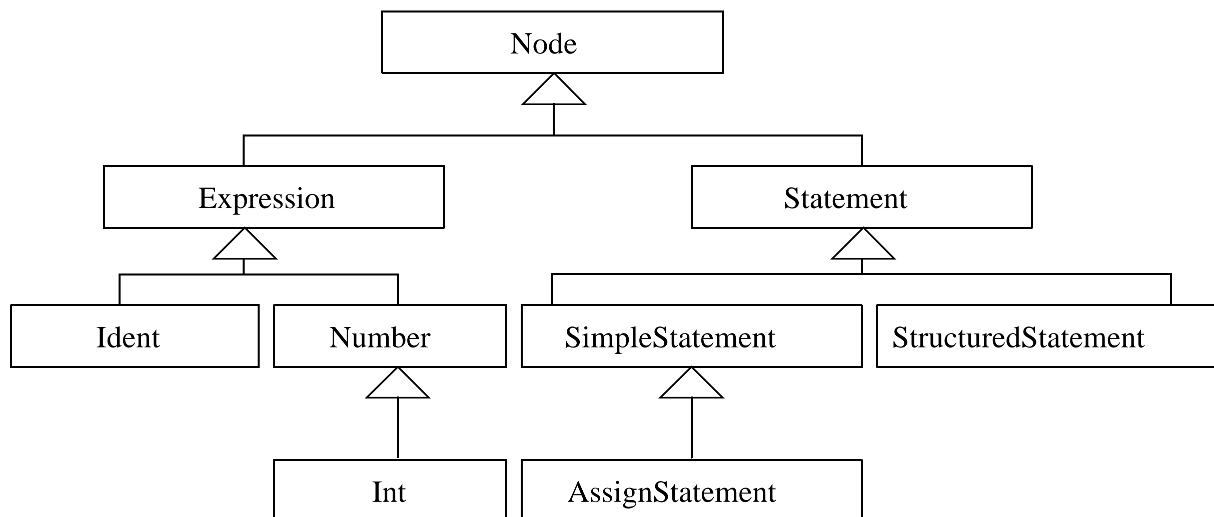
```
PROCEDURE ( VAR n : IntNode ) tp () : INTEGER;
BEGIN
    RETURN intTp;
END tp;
```

- Zugriff:

```
PROCEDURE ( VAR n : AssignNode ) analyze;
    …
    IF NOT compatible( n.lValue.tp(), n.rValue.tp() ) THEN …
```

Â Jeder Knotentyp kann die Methode selbst (re-) definieren.

Alternative 3: Bessere Strukturierung



Darstellung in Oberon-2

```
Node              =  RECORD
END;

ExpressionNode    =  RECORD ( Node )
   tp             :  INTEGER;
END;
IdentNode         =  RECORD ( ExpressionNode )
   repr           :  ARRAY OF CHAR;
END;
IntNode           =  RECORD ( ExpressionNode )
   repr           :  ARRAY OF CHAR;
END;
```

```
StatementNode            =  RECORD ( Node )
END;

SimpleStatementNode      =  RECORD ( StatementNode )
END;

AssignNode               =  RECORD ( SimpleStatementNode )
    lValue, rValue       :  ExpressionNode;
END;
```

Â Der Zugriff auf das Typfeld ist erlaubt:

```
PROCEDURE ( VAR n : AssignNode ) analyze;
    …
    IF NOT compatible( n.lValue.tp, n.rValue.tp ) THEN …
```


Überlegungen zur Strukturierung

- Bisher:

  Anweisungen und Ausdrücke aus Pascal betrachtet

- Nun:

  Ausdehnen auf andere imperative und objektorientierte Sprachen

- Voraussetzung:

  Imperative und objektorientierte Sprachen haben viele ähnliche Konzepte

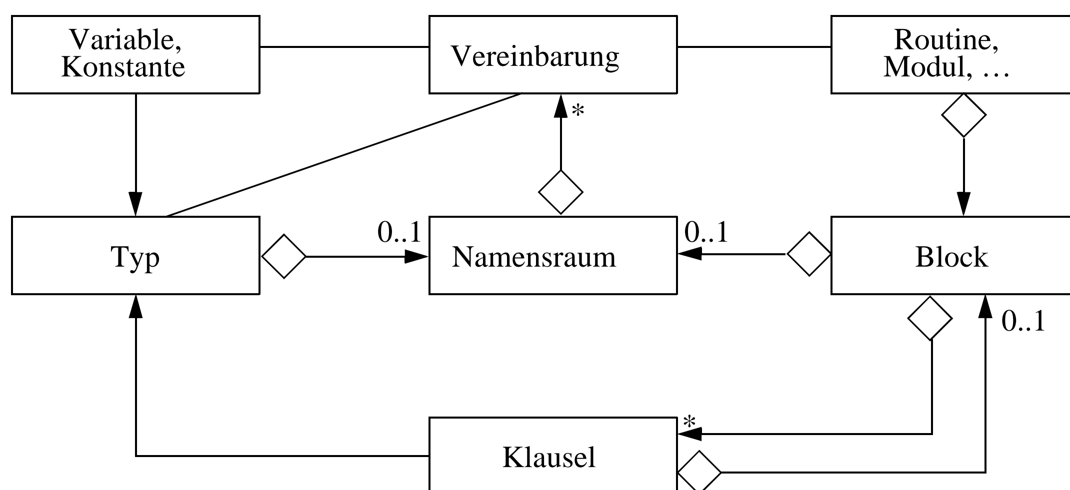
Systematik von Sprachkonzepten (1)

Untersuchte Sprachen:

- FORTRAN
- ALGOL 60, 68
- Common Lisp
- PL/I
- Simula 67, Smalltalk-80
- C, ANSI C, C++, Java
- Pascal, Modula-2, Oberon-2

Systematik von Sprachkonzepten (2)

Elementare Konzepte imperativer / objektorientierter Sprachen



Notation: UML

Systematik von Sprachkonzepten (3)

Kategorien von Sprachkonzepten:

- Vereinbarungen:
  Typen, Variablen, Routinen, …
- Klauseln:
  Anweisungen, Ausdrücke
- Blöcke:
  integrierendes Konstrukt
- Namensräume:
  an Blöcke gebunden

Ergebnis der Strukturierung …

- … ist eine Klassenanordnung, welche die Zusammenhänge von Sprachkonzepten repräsentiert.
- … ist eine statische Struktur.

Speziellere Konstrukte werden als Unterklassen allgemeinerer Konstrukte realisiert:

Spezialisierung = Beerbung

Ideen für eine Bibliothek mit Sprachkonzepten

o Die Bibliothek enthält allgemeine Formen von Sprachkonzepten

o Die Anpassung an eine konkrete Sprache erfolgt durch Beererbung

o Woher kommt die Dynamik (die Routinen für die Analyse und die Codegenerierung)?

‰ Beispiel in Oberon-2: Die Routinen werden als Methoden realisiert.

Wiederverwendung

o Wiederverwendung kann erreicht werden durch „Mitbenutzen“ der Methoden der Oberklasse.
Das kann (sollte) automatisch passieren.

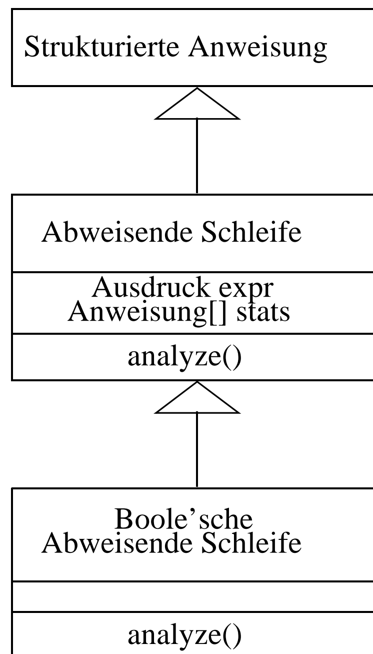
Â Das ist wegen der Spezialisierung von der Ober- zur Unterklasse immer möglich!

‰ Achtung: Die Instanziierung einiger Klassen muß verhindert werden!

  Â Oberon-2 bietet diese Möglichkeit nicht.

  Â Java besitzt die adäquaten Möglichkeiten: abstrakte Klassen.

Beispiel für Wiederverwendung: abweisende Schleife (WHILE)



Realisierung in Oberon-2

```
PROCEDURE ( VAR n : WhileStatementNode ) analyze;
VAR i : INTEGER;
BEGIN
   n.analyze^();
   n.expr.analyze();
   FOR i := 1 TO LEN( n.stats )-1 DO
      n.stats[i].analyze();
   END;
END analyze;
```

```
PROCEDURE ( VAR n : StatementNode ) analyze;
BEGIN
    (* ... *)
END analyze;

PROCEDURE ( VAR n : StructuredStatementNode ) analyze;
BEGIN
    n.analyze^();
    (* ... *)
END analyze;
```

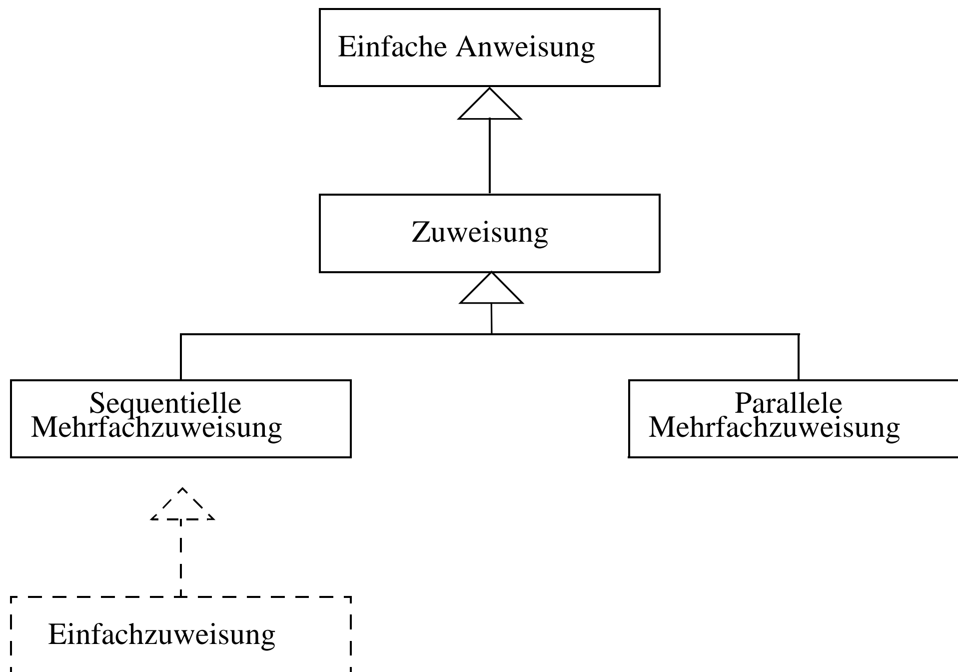

Spezialisierung der abweisenden Schleife

Es soll zusätzlich (!) geprüft werden, daß der Ausdruck vom Typ Boolean ist:

```
PROCEDURE ( VAR n : BooleanWhileStatNode ) analyze;
BEGIN
    n.analyze^();
    IF NOT (n.expr.tp = boolTp) THEN
        …
    END;
END analyze;
```

Â Nur die zusätzliche Überprüfung muß formuliert werden!

Beispiel für Wiederverwendung: Zuweisungen



Realisierung in Java: die Oberklassen

```
abstract class StatementNode extends Node
{
    void analyze ()
    { // …
    } // analyze
} // StatementNode

abstract class SimpleStatementNode extends StatementNode
{
    void analyze ()
    {   super.analyze();
        // …
    } // analyze
} // SimpleStatementNode
```

Realisierung in Java: die Mehrfachzuweisung

```
abstract class MultipleSequentialAssignNode
                                        extends SimpleStatementNode
{

    abstract ExpressionNode [] lValues ();

    abstract ExpressionNode [] rValues ();

    …
```



```
    …

    void analyze ()
    { int i;
        super.analyze();
        if ( lValues().length != rValues().length ) { … }
        for ( i=0; i<lValues().length; i++ ) {
            lValues()[i].analyze();
            rValues()[i].analyze();
            if ( !Type.compatible( lValues()[i].tp, rValues()[i].tp ) )
                { … }
        }
    } // analyze
} // MultipleSequentialAssignNode
```

Die konkrete Zuweisung aus Pascal , Grundlage: die Syntax

- Die Struktur der Klasse „AssignmentStatementNode“ kann direkt aus der EBNF-Darstellung der Produktion „AssignmentStatement“ abgeleitet werden:

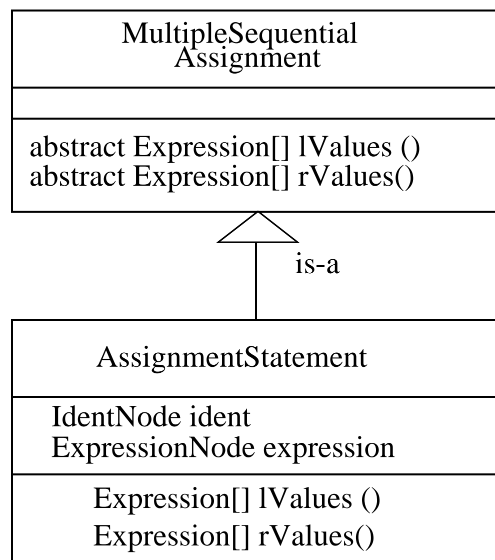
```
AssignmentStatement = Ident ":=" Expression.

class AssignmentStatementNode
{
    IdentNode ident;
    ExpressionNode expression;
} // AssignmentStatement
```

- Ebenfalls werden (automatisch) Klassen für „Ident“ und „Expression“ erzeugt:

Ident → IdentNode

Expression → ExpressionNode

Die Zuordnung Syntax (-klasse) → Bibliothek (-sklasse)

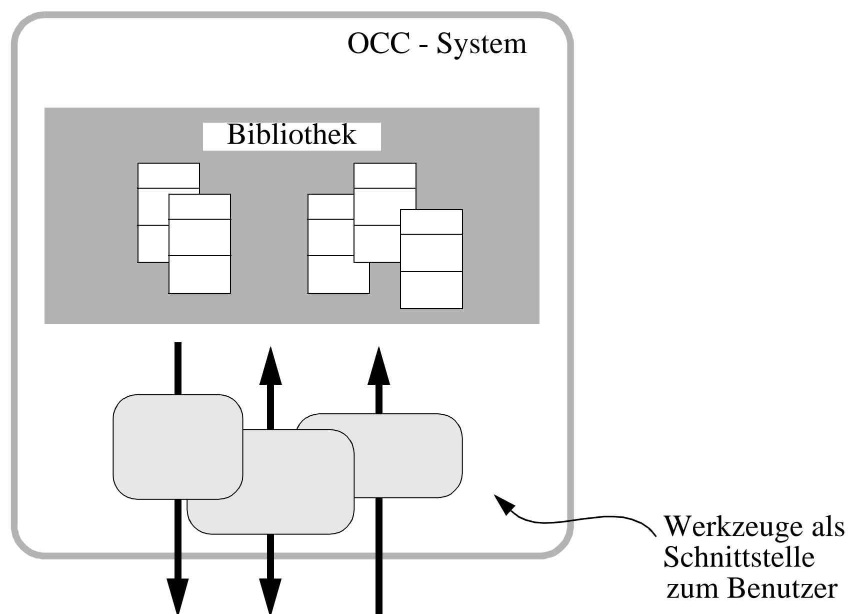


Â Aus dieser Zuordnung entsteht die Beerbungsbeziehung.

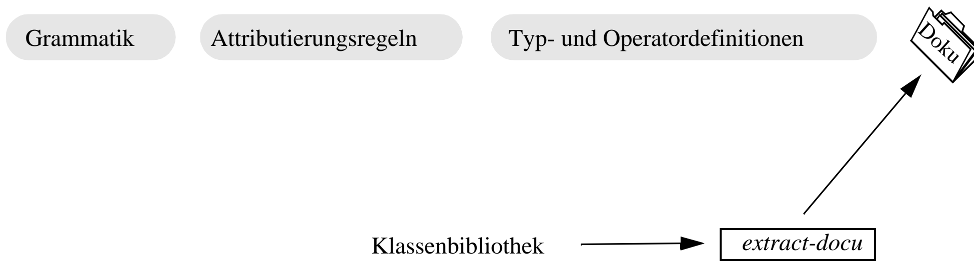
Realisierung in Java: Anpassen der Methoden

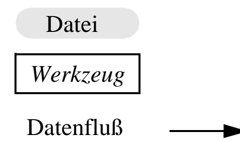
```
class AssignmentStatementNode extends MultipleSequentialAssignNode
{   IdentNode           ident;
    ExpressionNode      expression;

    ExpressionNode [] lValues ()
    {   ExpressionNode [] left = { ident };
        return left;
    } // lValues

    ExpressionNode [] rValues ()
    {   ExpressionNode [] right = { expression };
        return right;
    } // rValues
} // SingleAssignNode
```

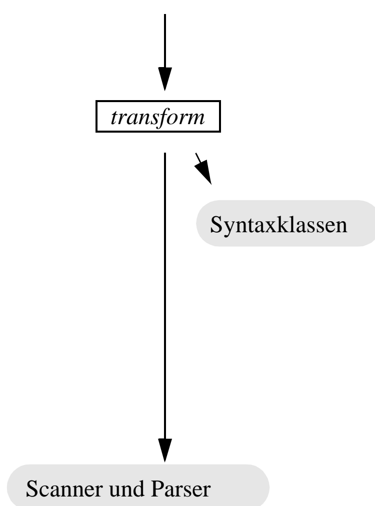

Objectoriented Compiler Construction

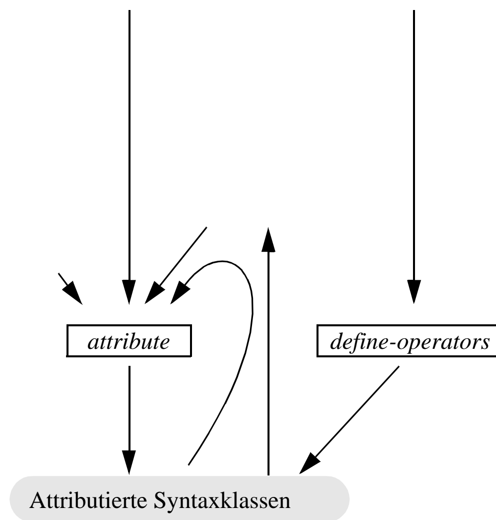


Der Einsatz des OCC-Systems





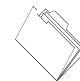


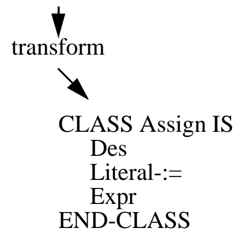


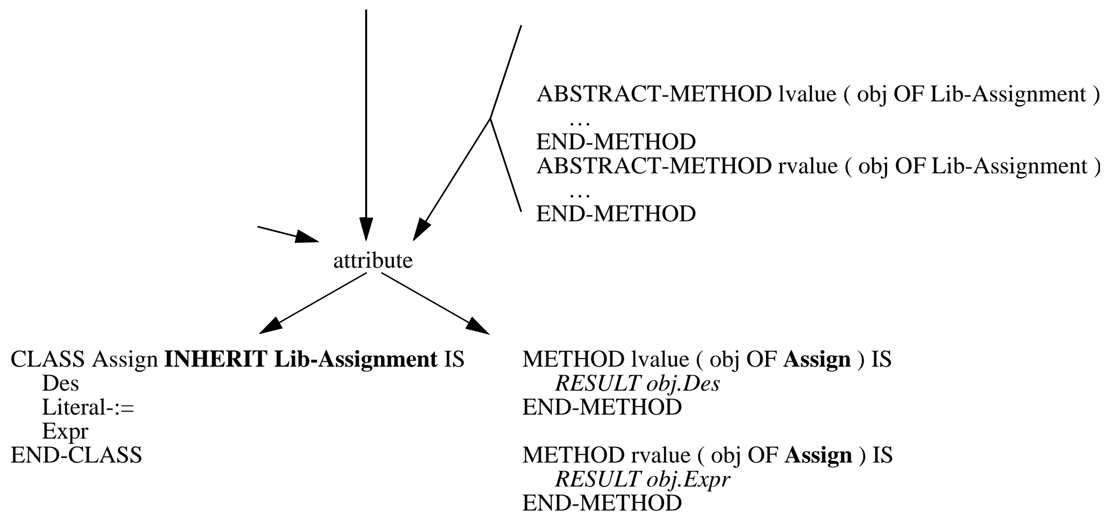
Beispiel für eine konkrete Anpassung

Assign = Des":=" Expr. *Assign IS-A Lib-Assignment.*

CLASS Lib-Assignment IS
…
END-CLASS




transform
CLASS Assign IS
Des
Literal-:=
Expr
END-CLASS


ABSTRACT-METHOD lvalue (obj OF Lib-Assignment)
...
END-METHOD
ABSTRACT-METHOD rvalue (obj OF Lib-Assignment)
...
END-METHOD
attribute
CLASS Assign INHERIT Lib-Assignment IS
Des
Literal-:=
Expr
END-CLASS
METHOD lvalue (obj OF Assign) IS
RESULT obj.Des
END-METHOD
METHOD rvalue (obj OF Assign) IS
RESULT obj.Expr
END-METHOD

Phasen der Übersetzung: lexikalische und syntaktische Analyse

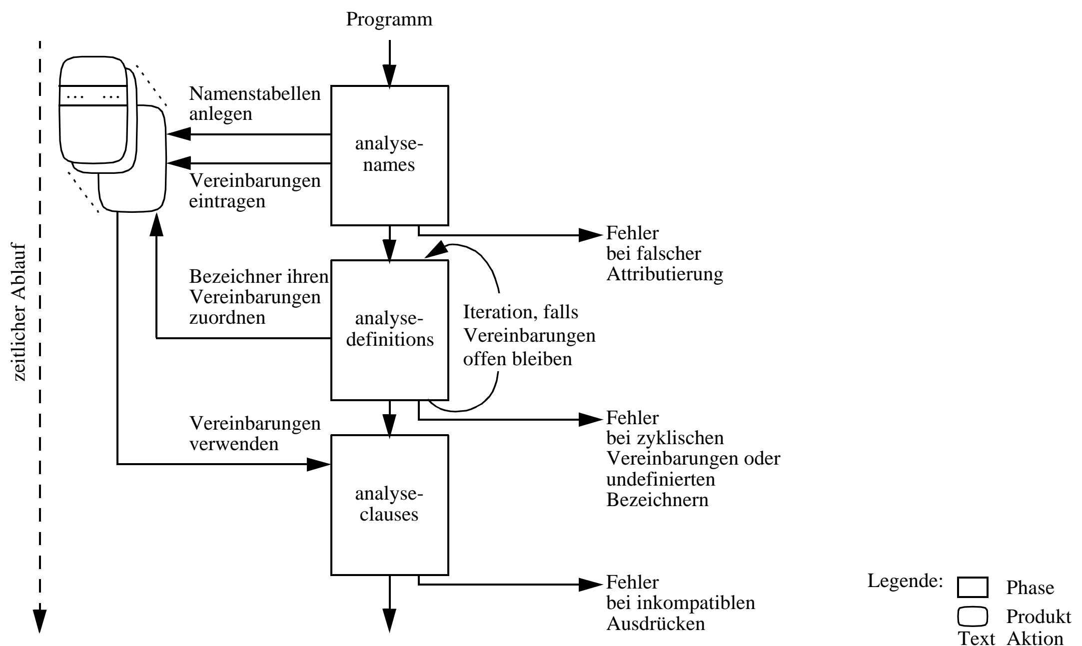
o Während der Syntaxanalyse:
  Es werden die Objekte erzeugt, auf denen die semantische Analyse und die Codegenerierung ablaufen.

Â Die lexikalische / syntaktische Analyse kann nicht durch Methoden realisiert werden.

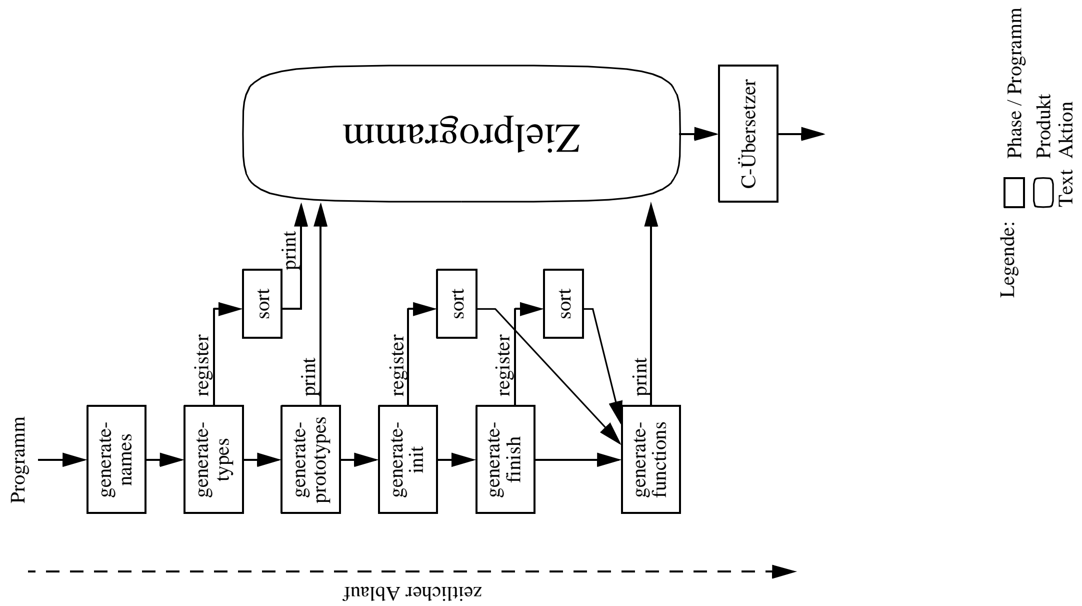
Â Man kann auf Standard-Werkzeuge zurückgreifen:
  Im OCC-System sind das flex und bison (lex und yacc).

‰ Erfahrung:
  Es ist günstig, einen möglichst mächtigen Parse-Algorithmus zu wählen!

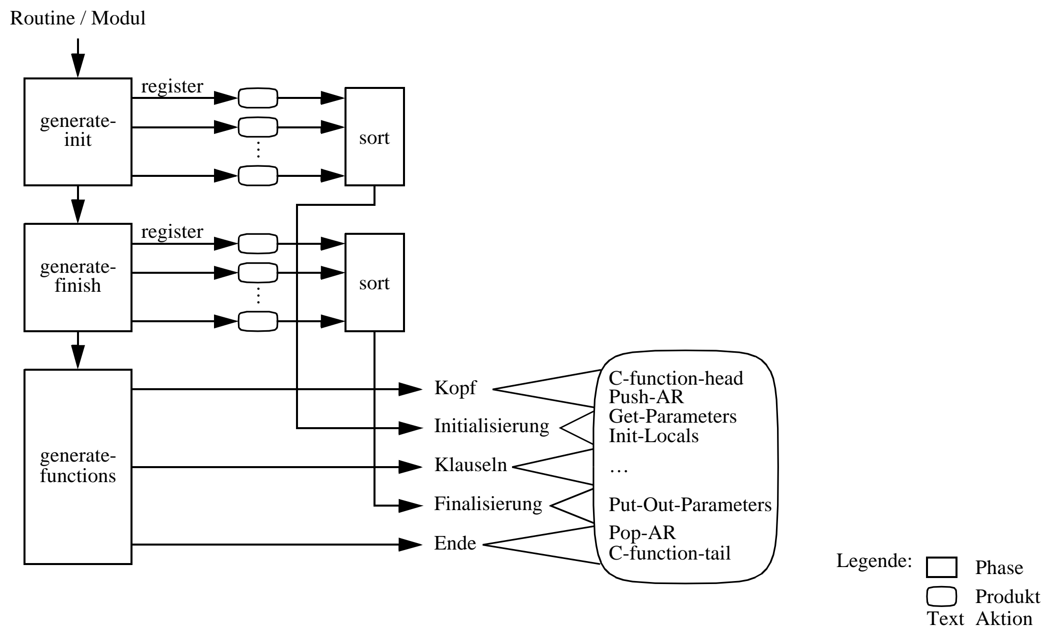
Phasen der Übersetzung: die semantische Analyse



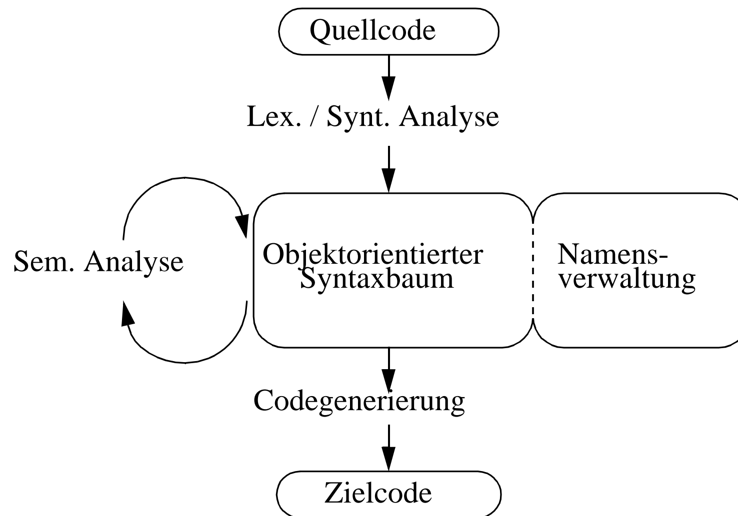
Phasen der Übersetzung: die Codegenerierung



Konstruktion von Blöcken im Zielcode



Zusammenfassung: Prinzip



Erfahrungen mit dem OCC-System (1)

o Die Komplexität eines Übersetzers korreliert mit der Komplexität der zu übersetzenden Sprache

o Viele Sprachen enthalten ähnliche Konzepte

‰ Wiederverwenden von Übersetzerkomponenten ist sinnvoll, um die Komplexität zu reduzieren

Erfahrungen mit dem OCC-System (2)

- Nachteile:
  - Erzeugte Übersetzer sind ineffizient
  - Benutzungsschnittstelle des Prototyps

- Vorteile:
  - Schnelle Entwicklung
  - Konzentration auf die neuen Konzepte
  - Konsistenzprüfung des Zwischencodes

Daten der prototypischen OCC-Implementierung

- Anzahl der Klassen: 360

- Anzahl der Nachrichten (versch. Methoden): 130

- Implementierungssprache: W-Lisp

- Dokumentation als Hypertext aufgebaut