

# Optimierende Compiler

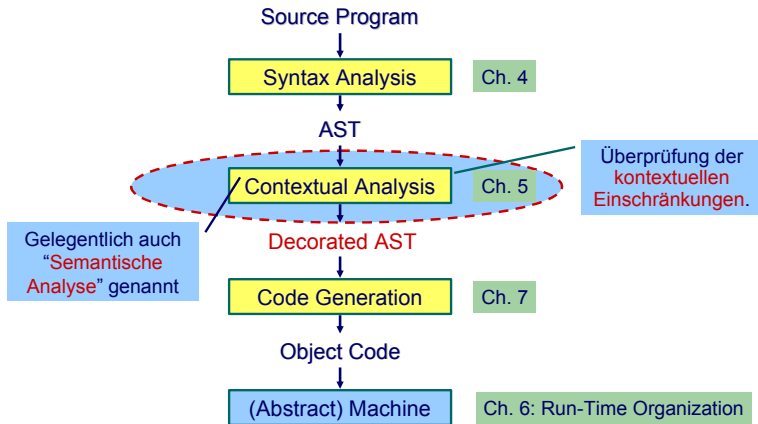
## 3. Kontextanalyse

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt

Sommersemester 2011

# Einleitung



# Kontextuelle Einschränkungen: Geltungsbereiche

```
let
  const m ~ 2;
  var n: Integer
in begin
  ..
  n := m*2;
  ..
end
```

Deklaration von n:  
Bindung

Benutzung von von n:  
Verwendung

??

```
let
  var n: Integer
in begin
  ..
  n := m*2;
end
```

Verwendung von m

Falls im Geltungsbereich der  
Verwendung vom m keine Bindung  
von m existiert: Fehler!

## Typen

- Jeder Wert hat einen Typ
- Jede Operation
  - ... hat Anforderungen an die Typen der Operanden
  - ... hat Regeln für den Typ des Ergebnisses

... auch nicht bei allen Programmiersprachen.

- Hier: statische Typisierung (zur Compile-Zeit)
- Alternativ: dynamische Typisierung (zur Laufzeit)

- Benutzung eines Bezeichners muss passende Deklaration haben
- Funktionsaufrufe müssen zu Funktionsdefinitionen passen
- LHS einer Zuweisung muss eine Variable sein
- Ausdruck in `if` oder `while` muß `Boolean` sein
- Beim Aufruf von Unterprogrammen müssen Anzahlen und Typen der aktuellen Parameter mit den formalen Parametern passen
- ...

- Bezeichner sind zunächst Zeichenketten
- Bekommen Bedeutung durch **Kontext**
  - Variablen, Konstanten, Funktion. . . .
- Bei jeder Benutzung nach Namen suchen
  - . . . viel zu **langsam**
- Besser: Weitgehende Vermeidung von String-Operationen
  - Nehme Zuordnung durch direktes Nachschlagen in Tabelle vor
  - Genannt: Symboltabelle, Identifizierungstabelle, . . .

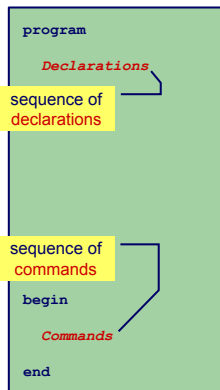
- Beispiel für zugeordnete Attribute
  - Typ int, char, boolean, record, array pointer, ...
  - Art Konstante, Variable, Funktion, Prozedur, Wert-Parameter, ...
  - Sichtbarkeit Public, private, protected
  - Anderes synchronized, static, volatile, ...
- Typische Operationen
- **Eintragen** einer neuen Zuordnung Namen-Attribute
- **Abrufen** der Attribute zu einem Namen
- Hierarchische Blockorganisation



- **Geltungsbereich** von Zuordnung von Namen zu Attributen innerhalb des Programmes
- **Block** Konstrukt im Programmtext zur Beschreibung von Geltungsbereichen
  - In Triangle:  
`let` Declarations `in` Commands  
`proc` P ( formal-parameters ) ~ Commands
  - In Java:  
Geltungsbereiche durch {, } gekennzeichnet
- Unterschiedliche Handhabungsmöglichkeiten von Geltungsbereichen

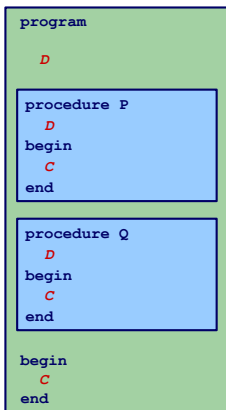
# Geltungsbereiche und Symboltabellen

- Charakteristika
  - Nur **ein** Block
  - Alle Deklarationen gelten **global**
- Regeln für Geltungsbereiche
  - Bezeichner darf nur genau **einmal** deklariert werden
  - Jeder benutzte Bezeichner **muß** deklariert sein
- Symboltabelle
  - Für jeden Bezeichner genau **ein** Eintrag in der Symboltabelle
  - Abruf von Daten muß schnell gehen (binärer Suchbaum, Hash-Tabelle)
- Beispiele: BASIC, COBOL, Skriptsprachen



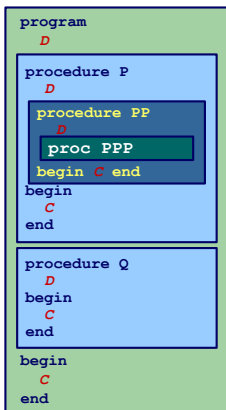
```
public class Attribute {  
    // Attribute details  
    ...  
}  
  
public class IdentificationTable {  
  
    /** Adds a new entry */  
    public void enter(String id, Attribute attr) { ... }  
  
    /** Retrieve a previously added entry. Returns null  
        when no entry for this identifier is found */  
    public Attribute retrieve(String id) { ... }  
  
    ...  
}
```

- Charakteristika
  - Mehrere überlappungsfreie Blöcke
  - Zwei Geltungsbereiche: Global und Lokal
- Regeln für Geltungsbereiche
  - Global deklarierte Bezeichner dürfen nicht global redefiniert werden
  - Lokal deklarierte Bezeichner dürfen nicht im selben Block redefiniert werden
  - Jeder benutzte Bezeichner muss global oder lokal zu seiner Verwendungsstelle deklariert sein
- Symboltabelle
  - Bis zu zwei Einträge für jeden Bezeichner (global und lokal)
  - Nach Bearbeiten eines Blocks müssen lokale Deklarationen verworfen werden
- Beispiel: FORTRAN



```
public class IdentificationTable {  
  
    /** Adds a new entry */  
    public void enter(String id, Attribute attr) { ... }  
  
    /** Retrieve a previously added entry. If both global and local entries exist  
        for id, return the attribute for the local one. Returns null  
        when no entry for this identifier is found */  
    public Attribute retrieve(String id) { ... }  
  
    /** Add a local scope level to the table, with no initial entries */  
    public void openScope() { ... }  
  
    /** Remove the local scope level from the table.  
        Deletes all entries associated with it */  
    public void closeScope() { ... }  
  
    ...  
}
```

- Charakteristika
  - Blöcke ineinander **verschachtelt**
  - Beliebige Schachtelungstiefe der Blöcke
- Regeln für Geltungsbereiche
  - Kein Bezeichner darf mehr als einmal innerhalb eines Blocks deklariert werden
  - Kein Bezeichner darf verwendet werden, ohne dass er lokal oder in den **umschliessenden** Blöcken deklariert wurde
- Symboltabelle
  - **Mehrere** Einträge je Bezeichner möglich
  - Aber maximal ein Paar (Verschachtelungstiefe, Bezeichner)
  - Schneller Abruf des Eintrags mit der größten Verschachtelungstiefe
- Beispiele: Pascal, Modula, Ada, Java, ...



# Beispiel: Verschachtelte Blockstruktur

```
let !level 1
  var a, b, c ;
in begin
  let !level 2
    var a, b ;
  in begin
    let !level 3
      var a, c ;
    in begin
      a := b + c ;
    end;
    a := b + c ;
  end;
  a := b + c ;
end
```

Geltungsbereiche  
und Sichtbarkeit

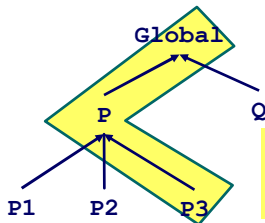
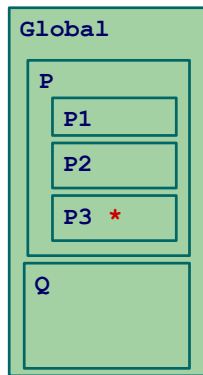
```
let !level 1
  var a, b, c ;
in begin
  let !level 2
```

Geltungsbereiche  
und Sichtbarkeit



# Struktur der Geltungsbereiche

- Für Sprachen mit verschachtelter Blockstruktur
- Modellierung als Baum



Suchpfad für ein  
verwendendes  
Auftreten in P3

Während der Programmanalyse ist immer  
nur ein **einzelner** Pfad sichtbar.

```
public class IdentificationTable {  
  
    /** Adds a new entry */  
    public void enter(String id, Attribute attr) { ... }  
  
    /** Retrieve a previously added entry with the deepest scope level.  
        Returns null when no entry for this identifier is found */  
    public Attribute retrieve(String id) { ... }  
  
    /** Add a new deepest scope level to the table, with no initial entries */  
    public void openScope() { ... }  
  
    /** Remove the deepest local scope level from the table.  
        Deletes all entries associated with it */  
    public void closeScope() { ... }  
  
    ...  
}
```

- Verschiedene Varianten
  - Verkettete Liste und lineare Suche
    - Einfach aber langsam
    - In Triangle verwendet (natürlich ...)
  - Hier: Bessere Möglichkeiten
  - Hash-Tabelle (effizienter)
  - **Stack** aus Hash-Tabellen
- Design-Kriterium
  - **Gleiche** Bezeichner tauchen häufiger in Tabelle auf
  - Aber auf unterschiedlichen **Ebenen**
  - Abgerufen wird immer der am **tiefsten** gelegene

```
public class SymbolTable {  
    private Map    symtab      = new HashMap();  
    private Stack scopeStack = new Stack();  
    ...  
}
```

Optimiert Schließen eines Geltungsbereiches

In Java 5 (aka 1.5):

```
Map<String, Stack<Attribute>> symtab;  
Stack<List<String>> scopeStack;
```

## `syntab`

- Bildet von `Strings` auf `Attribute`-Objekte ab
- Bezeichnernamen dienen als **Schlüssel**
- **Wert** ist ein Stack aus Attributen, **obenauf** liegt die Deklaration mit der **tiefsten** Verschachtelungsebene

## `scopeStack`

- Stack bestehend aus Listen von Strings
- Bei **Öffnen** eines neuen Geltungsbereichs:
  - Lege leere Liste auf `scopeStack`
  - Jeder in diesem Bereich gefundene Bezeichner wird in Liste eingetragen
- Bei **Schließen** des aktuellen Geltungsbereiches
  - Gehe Liste oben auf `scopeStack` durch
  - Lösche alle diese Bezeichner aus `syntab` (entferne jeweils oberstes Stapелеlement)
  - Entferne dann oberstes Elements von `scopeStack`

Andere Implementierungen möglich!

# Attribute

- Welche **Informationen** konkret zu einem Bezeichner speichern?
- **Wofür** werden Attribute gebraucht?
- Mindestens für
  - Überprüfung der Regeln für Geltungsbereiche von Deklarationen
    - Bei geeigneter Implementierung der Symboltabelle: Einfaches Abrufen reicht
    - Alle Regeln bereits in Datenstruktur realisiert
  - Überprüfung der Typregeln
    - Erfordert Abspeicherung von **Typinformationen**
  - (Code-Erzeugung)
    - Benötigt später z.B. **Adresse** der Variable im Speicher



## Beispiel 1:

```
let const m~2;  
in m + x
```

## Beispiel 2:

```
let const m~2 ;  
    var n:Boolean  
in begin  
    n := m<4;  
    n := n+1  
end
```

## Beispiele: Verwendung von Attributen 2

### Beispiel 1:

```
let const m~2;  
in m + x Undefiniert!
```



Geltungsbereichs-  
regeln

### Beispiel 2:

```
let const m~2 ;  
  var n:Boolean  
in begin  
  n := m<4;  
  n := n+1 Typfehler!  
end
```



Typregeln

## Imperativer Ansatz (explizite Speicherung)

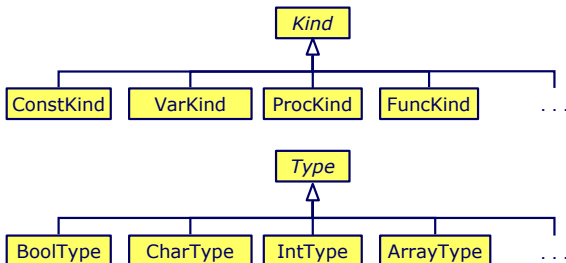
```
public class Attribute {  
  
    public static final byte // kind  
        CONST = 0,  
        VAR   = 1,  
        PROC  = 2,  
        ... ;  
  
    public static final byte // type  
        BOOL  = 0,  
        CHAR  = 1,  
        INT   = 2,  
        ARRAY = 3,  
        ... ;  
  
    public byte kind;  
    public byte type;  
}
```

OK für sehr einfache  
Sprachen

## Objektorientierter Ansatz (explizite Speicherung)

```
public class Attribute {  
    public Kind kind;  
    public Type type;  
}
```

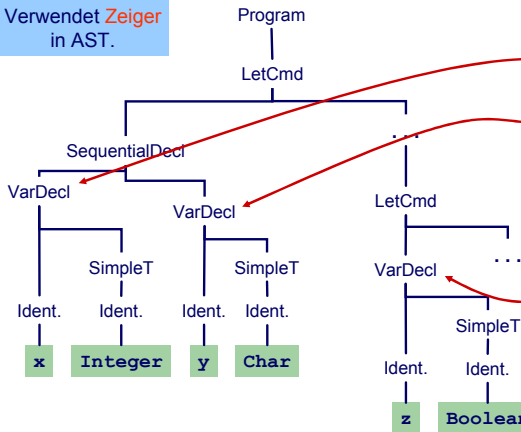
Funktioniert, wird aber bei  
**realistischer** Sprache  
sehr leicht **unhandlich**



- Schon bloße Aufzählung in Form von Klassen langatmig
- Noch nicht berücksichtigt: Kombinationen
  - `array [1:10] of record int x; char y end;`
- Explizite Strukturen können leicht sehr **komplex** werden
  
- **Idee:** Im AST stehen bereits alle Daten
  - Deklarations-Unterbaum
- Als Attribute einfach Verweise auf **ursprüngliche Definition** eintragen
  - Dabei Geltungsbereiche beachten!

# AST-basierte Attribute

Verwendet **Zeiger**  
in AST.



```
let var x: Integer;
    var y: Char
in begin
    ...
    let var z: Boolean
    in ...
end
```

level	id	Attr.
1	x	•
1	y	•
2	z	•

Sehr hilfreich für  
**Dekoration** des  
ASTs.

# Identifikation

- Erster Schritt der Kontextanalyse
- Beinhaltet Aufbau einer geeigneten Symboltabelle
- Aufgabe: Ordne Verwendungen von Bezeichnern ihren Definitionen zu
- Durch Pass über den AST realisierbar ...
  
- aber besser: Kombinieren mit nächstem Schritt

➔ **Typprüfung**



# Typprüfung

- **Was** ist ein **Typ**?
  - “Eine Einschränkung der möglichen Interpretationen eines Speicherbereiches oder eines anderen Programmkonstrukts.”
  - Eine Menge von Werten
- **Warum** Typen benutzen?
  - **Fehlervermeidung**: Verhindere eine Art von Programmierfehlern (“eckiger Kreis”)
  - **Laufzeitoptimierung**: Bindung zur Compile-Zeit erspart Entscheidungen zur Laufzeit
- **Muß** man immer Typen verwenden?
  - **Nein**, viele Sprachen kommen ohne aus
    - Assembler, Skriptsprachen, LISP, ...

- Bei **statischer Typisierung** ist jeder Ausdruck  $E$  **entweder**
  - Misstypisiert, **oder**
  - Hat einen statischen Typ  $T$ , der ohne Evaluation von  $E$  bestimmt werden kann
- $E$  wird bei jeder (fehlerfreien) Evaluation den statischen Typ  $T$  haben
- Viele moderne Programmiersprachen bauen auf statische Typüberprüfung auf
  - OO-Sprachen haben aber auch dynamische Typprüfungen zur Laufzeit (Polymorphismus)

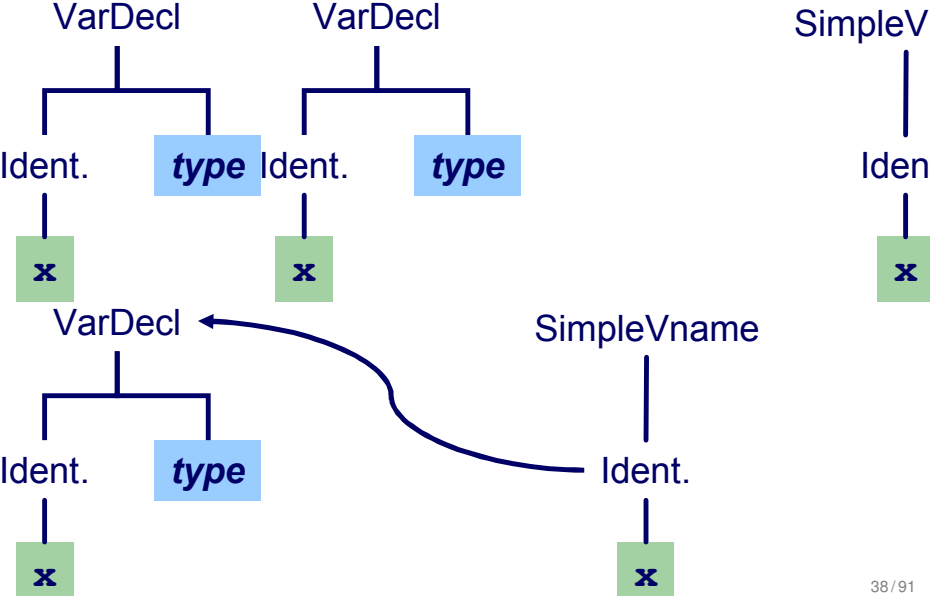
## Generelles Vorgehen

- 1 Berechne oder leite Typen von Ausdrücken her
  - Aus den Typen der Teilausdrücke und der Art der Verknüpfung
- 2 Überprüfe, das Typen der Ausdrücke Anforderungen aus dem Kontext genügen
  - Beispiel: Bedingung in **if/then** muß einen Boolean liefern

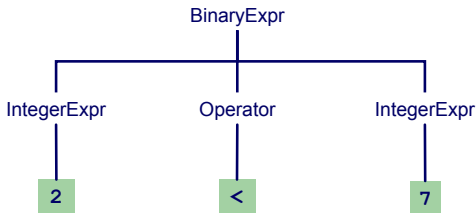
## Genauer: Bottom-Up Verfahren für statisch typisierte Programmiersprache

- Typen an den **Blättern** des AST sind bekannt
  - Literale** Direkt aus Knoten (true/false, 23, 42, 'a')
  - Variablen** Aus Symboltabelle
  - Konstanten** Aus Symboltabelle
- Typen der internen Knoten herleitbar aus
  - Typen der Kinder
  - **Typregel** für die Art der Verknüpfung im Ausdruck

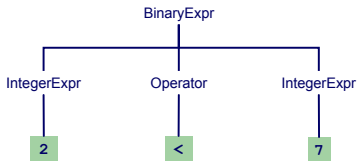
# Beispiel: Typherleitung für Variablen



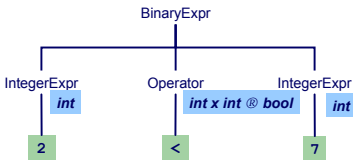
# Beispiel: Typherleitung für Ausdrücke



**Typregel für Binären Ausdruck:**  
Wenn  $op$  Operation vom Typ  $T_1 \times T_2 \rightarrow R$  ist, dann ist  $E_1$  op  $E_2$  typkorrekt und vom Typ  $R$  wenn  $E_1$  and  $E_2$  typkorrekt sind und typkompatibel zu  $T_1$  bzw.  $T_2$  sind



**Typregel für Binären Ausdruck:**  
Wenn  $op$  Operation vom Typ  $T_1 \times T_2 \rightarrow R$  ist, dann ist  $E_1$  op  $E_2$  typkorrekt und vom Typ  $R$  wenn  $E_1$  and  $E_2$  typkorrekt sind und typkompatibel zu  $T_1$  bzw.  $T_2$  sind



Anweisungen mit Ausdrücken

Typregel für **ifCommand**:

`if E then C1 else C2`

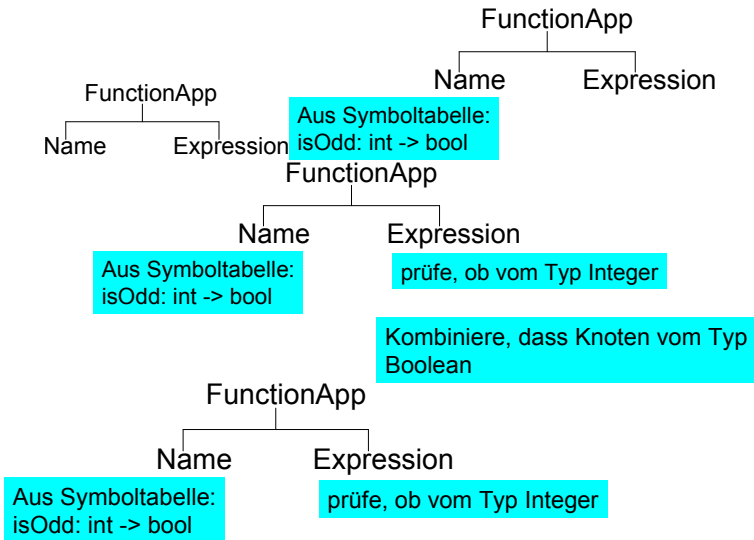
ist **typkorrekt** genau dann, wenn

- *E* vom Typ Boolean ist und
- *C1* und *C2* selbst typkorrekt sind



# Beispiel: Typherleitung für Funktionsaufruf

`isOdd(42)`



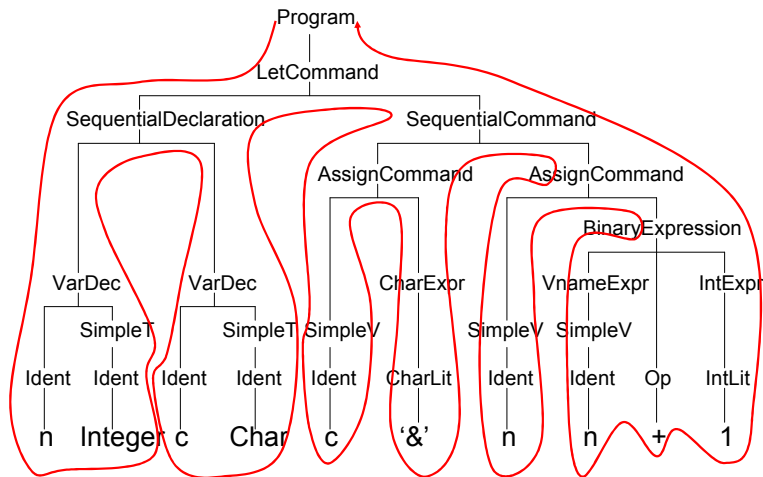
```
func f ( x : ParamType ) : ResultType ~  
Expression
```

- Typprüfung des Körpers **Expression**
- Stelle sicher, dass Ergebnis von **ResultType** ist
- Dann Herleitung: **f: ParamType → ResultType**

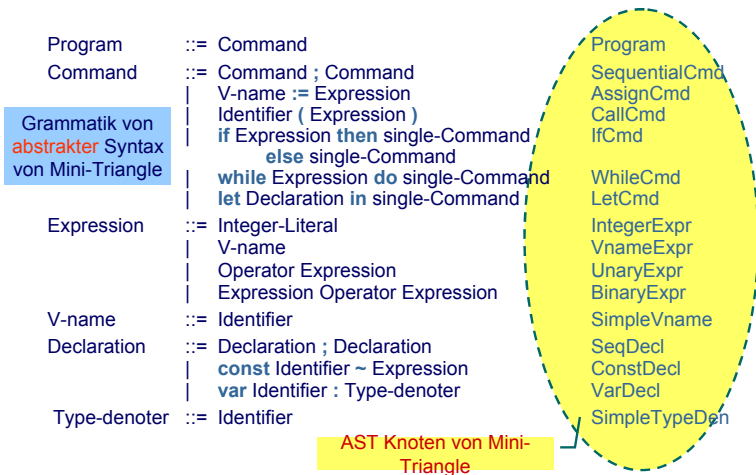
Idee: Vereinheitliche Typüberprüfung von Funktionen und Operatoren

- **+**: **Integer × Integer → Integer**
- **<**: **Integer × Integer → Boolean**

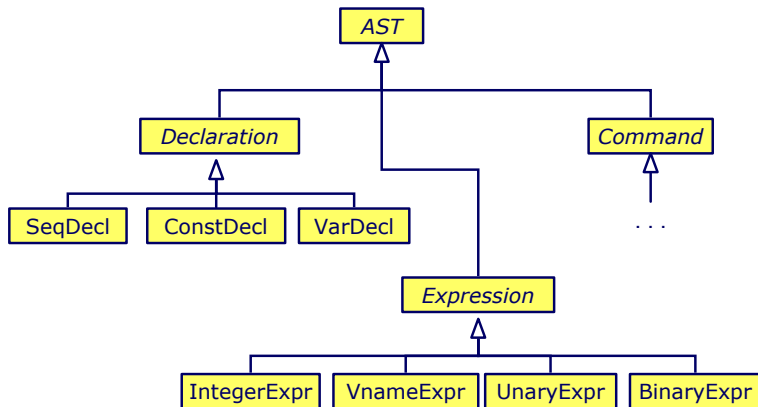
- Kombiniere Identifikation und Typprüfung in **einem** Pass
- Funktioniert, solange Bindung immer vor Verwendung
  - In (mini-)Triangle der Fall
- Mögliche Vorgehensweise
  - Tiefensuche von **links nach rechts** durch AST
  - Dabei sowohl Identifikation und Typüberprüfung
  - Speichere Ergebnisse durch **Dekorieren** des ASTs
    - Hinzufügen weiterer Informationen



# Abstrakter Syntaxbaum



# Klassenstruktur für AST



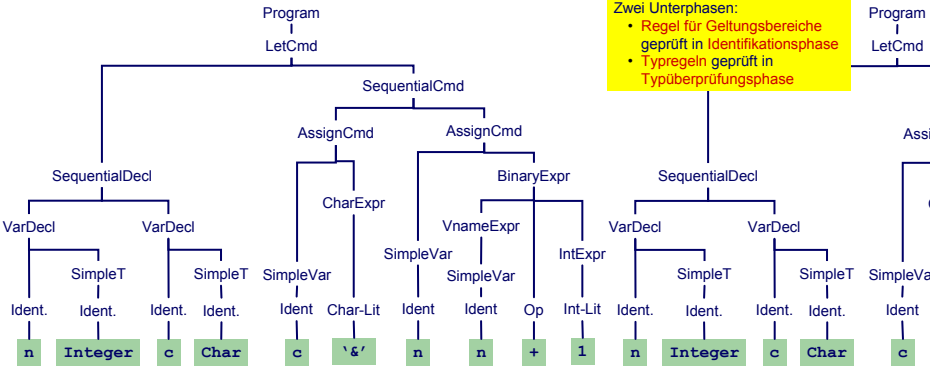
Expression ::= Integer-Literal	IntegerExpr
V-name	VnameExpr
Operator Expression	UnaryExpr
Expression Operator Expression	BinaryExpr

```
public class BinaryExpr extends Expression {
    public Expression E1, E2;
    public Operator O;
}

public class UnaryExpr extends Expression {
    public Expression E;
    public Operator O;
}

...
```

# Gewünschtes Ergebnis



Zwei Unterphasen:  
 • Regel für Geltungsbereiche geprüft in Identifikationsphase  
 • Typregeln geprüft in Typüberprüfungsphase

Zwei Unterphasen:  
 • Regel für Geltungsbereiche geprüft in Identifikationsphase  
 • Typregeln geprüft in Typüberprüfungsphase





Benötigt Erweiterung einiger AST Knoten um zusätzlich Instanzvariablen.

```
public abstract class Expression extends AST {  
    // Every expression has a type  
    public Type type;  
    ...  
}
```

```
public class Identifier extends Token {  
    // Binding occurrence of this identifier  
    public Declaration decl;  
    ...  
}
```

Wie nun bei Implementierung vorgehen?

# Implementierung

# 1. Versuch: Dekoration mit OO-Ansatz

- Erweitere jede AST-Subklasse um **Methoden** für
  - Typprüfung, Code-Erzeugung, Pretty-Printing, ...
- In jeder Methode: Durchlauf über Kinder

```
public abstract AST() {
    public abstract Object check(Object arg);
    public abstract Object encode(Object arg);
    public abstract Object prettyPrint(Object arg);
}
...
Program program;
program.check(null);
```

```
public abstract AST() {
    public abstract Object check(Object arg);
    public abstract Object encode(Object arg);
    public abstract Object prettyPrint(Object arg);
}
...
Program program;
program.check(null);
```

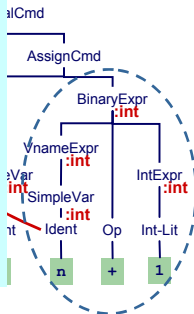
Rückgab

- **Vorteil** OO-Vorgehen leicht verständlich und implementierbar
- **Nachteil** Verhalten (Prüfung, Erzeugung, ...) ist **verteilt** über alle AST-Klassen, nicht sonderlich **modular**.

# Beispiel: Dekorierung via OO Ansatz

```
public abstract class Expression extends AST {
    public Type type;
    ...
}
public class BinaryExpr extends Expression {
    public Expression E1, E2;
    public Operator O;

    public Object check(Object arg) {
        Type t1 = (Type) E1.check(null);
        Type t2 = (Type) E2.check(null);
        Op op = (Op) O.check(null);
        Type result = op.compatible(t1,t2);
        if (result == null)
            report type error
        return result;
    }
    ...
}
```



```
public abstract class Expression extends AST {
    public Type type;
    ...
}
public class BinaryExpr extends Expression {
    public Expression E1, E2;
    public Operator O;

    public Object check(Object arg) {
        Type t1 = (Type) E1.check(null);
        Type t2 = (Type) E2.check(null);
        Op op = (Op) O.check(null);
        Type result = op.compatible(t1,t2);
        if (result == null)
            report type error
        return result;
    }
    ...
}
```

```
Object[] tmp = new Object[2];
tmp[0] = t1; tmp[1] = t2;
Type result = (Type) O.ch
```

## 2. Versuch: "Funktionaler" Ansatz

Besser (?): Hier alles Verhalten zusammen in einer Methode

```
Type check(Expr e) {
  if (e instanceof IntLitExpr)
    return representation of type int
  else if (e instanceof BoolLitExpr)
    return representation of type bool
  else if (e instanceof EqExpr) {
    Type t = check(((EqExpr)e).left);
    Type u = check(((EqExpr)e).right);
    if (t == representation of type int &&
        u == representation of type int)
      return representation of type bool
    ...
  }
```

➡ Nicht sonderlich OO, ignoriert eingebauten Dispatcher

- Engl. *Visitor Pattern*
- 1994 Gamma, Johnson, Helm, Vlissides (GoF)
- Neue Operationen auf Teilelementen (**part-of**) eines Objekts (z.B. AST)
- ... **ohne** Änderung der Klassen der Objekte
- Besonders nützlich wenn
  - viele unterschiedliche und
  - unzusammenhängende Operationen
- ... ausgeführt werden müssen
- **ohne** die Klassen der Teilelemente aufzublähen

- Operationen können mit dem Visitor-Pattern leicht **hinzugefügt** werden
- Visitor sammelt zusammengehörige Operationen und trennt sie von unverwandten
- Visitor durchbricht Kapselung
- Parameter und Return-Typen müssen in allen Visitors gleich sein
- Hängt stark von Klassenstruktur ab
- ... Visitor problematisch, wenn die Struktur sich noch ändert

- Definiere `visitor`-Schnittstelle für Besuch von AST-Knoten
- Füge zu jeder AST-Subklasse `xyz` eine einzelne `visit`-Methode hinzu
  - In der Literatur auch `accept` genannt, hier mißverständlich mit Parser
- Rufe dort Methode `visitXYZ` der `visitor`-Klasse auf

```
public abstract AST() {
    public abstract Object visit(Visitor v, Object arg);
}
public class AssignCmd extends Command {
    public Object visit(Visitor v, Object arg) {
        return v.visitAssignCmd(this, arg);
    }
}
```

```
public abstract AST() {
    public abstract Object visit(Visitor v, Object arg);
}
public class AssignCmd extends Command {
    public Object visit(Visitor v, Object arg) {
        return v.visitAssignCmd(this, arg);
    }
}
```

Unterschiedliche  
Methode realisieren  
(Typüberprüfung)



```
public class XYZ extends ... {  
    Object visit(Visitor v, Object arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```

```
public interface Visitor {  
    public Object visitProgram  
        (Program prog, Object arg);  
    ...  
    public Object visitAssignCmd  
        (AssignCmd cmd, Object arg);  
    public Object visitSequentialCmd  
        (SequentialCmd cmd, Object arg);  
    ...  
    public Object visitVnameExpression  
        (VnameExpression e, Object arg);  
    public Object visitBinaryExpression  
        (BinaryExpression e, Object arg);  
    ...  
}
```

Interface Visitor definiert `visitXYZ` für alle  
Subklassen `XYZ` von AST

```
public Object visitXYZ  
    (XYZ x, Object arg);
```

Jetzt alle benötigten Methoden zusammen in einer Klasse

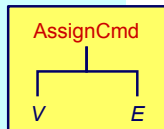
```
public class Checker implements Visitor {  
  
    private SymbolTable symtab;  
  
    public void check(Program prog) {  
        symtab = new SymbolTable();  
        prog.visit(this, null);  
    }  
  
    ... + implementations of all methods of Visitor  
}
```

Wurzelknoten des AST

# Beispiel: AssignCmd

```
public class XYZ extends ... {  
    Object visit(Visitor v,  
                Object arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```

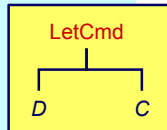
```
public Object visitAssignCmd  
    (AssignCmd com, Object arg) {  
    Type vType = (Type) com.V.visit(this, null);  
    Type eType = (Type) com.E.visit(this, null);  
    if (! com.V.variable)  
        error: left side is not a variable  
    if (! eType.equals(vType))  
        error: types are not equivalent  
    return null;  
}
```



# Beispiel: LetCmd

```
public class XYZ extends ... {  
    Object visit(Visitor v,  
                Object arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```

```
public Object visitLetCmd  
    (LetCmd com, Object arg) {  
    symtab.openScope();  
    com.D.visit(this, null);  
    com.C.visit(this, null);  
    symtab.closeScope();  
    return null;  
}
```

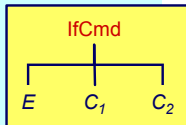


letCmd **öffnet** (und **schließt**) eine Ebene von Geltungsbereichen in **Symbole**ttabelle.

# Beispiel: IfCmd

```
public class XYZ extends ... {  
    Object visit(Visitor v,  
                Object arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```

```
public Object visitIfCmd  
    (IfCmd com, Object arg) {  
    Type eType = (Type)com.E.visit(this, null);  
    if (! eType.equals(Type.bool))  
        error: condition is not a boolean  
    com.C1.visit(this, null);  
    com.C2.visit(this, null);  
    return null;  
}
```



# Beispiel: IntegerExpr

```
public class XYZ extends ... {  
    Object visit(Visitor v,  
                Object arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```

```
public Object visitIntegerExpr  
    (IntegerExpr expr, Object arg) {  
    expr.type = Type.int;  
    return expr.type;  
}
```

Dekoriere den IntegerExpr  
Knoten im AST

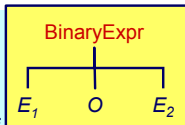
IntegerExpr

|  
IL

nicht ins Terminal  
IL absteigen

# Beispiel: BinaryExpr

```
public Object visitBinaryExpr
    (BinaryExpr expr, Object arg) {
    Type e1Type = (Type) expr.E1.visit(this, null);
    Type e2Type = (Type) expr.E2.visit(this, null);
    OperatorDeclaration opdecl =
        (OperatorDeclaration) expr.O.visit(this, null);
    if (opdecl == null) {
        error: no such operator
        expr.type = Type.error;
    } else if (opdecl instanceof BinaryOperatorDeclaration) {
        BinaryOperatorDeclaration bopdecl =
            (BinaryOperatorDeclaration) opdecl;
        if (! e1Type.equals(bopdecl.operand1Type))
            error: left operand has the wrong type
        if (! e2Type.equals(bopdecl.operand2Type))
            error: right operand has the wrong type
        expr.type = bopdecl.resultType;
    } else {
        error: operator is not a binary operator
        expr.type = Type.error;
    }
    return expr.type;
}
```



Weitere Methoden in  
PLPJ.

# Beispiel: **VarDecl** und **ConstDecl**

## // Declaration checking

```
public Object visitVarDeclaration (VarDeclaration decl, Object arg) {  
    decl.T.visit(this, null);  
    idTable.enter(decl.l.spelling, decl);  
    return null;  
}
```

```
public Object visitConstDeclaration (ConstDeclaration decl, Object arg) {  
    decl.E.visit(this, null);  
    idTable.enter(decl.l.spelling, decl);  
    return null;  
}
```

...



# Beispiel: simpleVName

```
// VName checking
public Object visitSimpleVName (SimpleVname vname, Object arg) {
    Declaration decl = vname.l.visit(this, null);
    if (decl==null) {
        // error: VName not declared
    } else if (decl instanceof ConstDeclaration) {
        vname.type = ((ConstDeclaration) decl).E.type;
        vname.variable = false;
    } else if (decl instanceof VarDeclaration) {
        vname.type = ((VarDeclaration) decl).T.type;
        vname.variable = true;
    }
    return vname.type;
}
```

# Zusammenfassung aller `visitXYZ`-Methoden

Program	<code>visitProgram</code>	<ul style="list-style-type: none"><li>• return <code>null</code></li></ul>
Command	<code>visit..Cmd</code>	<ul style="list-style-type: none"><li>• return <code>null</code></li></ul>
Expression	<code>visit..Expr</code>	<ul style="list-style-type: none"><li>• <b>dekoriere</b> ihn mit seinem <code>Typ</code></li><li>• return <code>Typ</code></li></ul>
Vname	<code>visitSimpleVname</code>	<ul style="list-style-type: none"><li>• <b>dekoriere</b> ihn mit seinem <code>Typ</code></li><li>• setze <code>Flag</code>, falls <code>Variable</code></li><li>• return <code>Typ</code></li></ul>
Declaration	<code>visit..Decl</code>	<ul style="list-style-type: none"><li>• trage alle <b>deklarierten</b> Bezeichner in <code>Symboltabelle</code> ein</li><li>• return <code>null</code></li></ul>
TypeDenoter	<code>visit..TypeDenoter</code>	<ul style="list-style-type: none"><li>• <b>dekoriere</b> ihn mit seinem <code>Typ</code></li><li>• return <code>Typ</code></li></ul>
Identifizier	<code>visitIdentifizier</code>	<ul style="list-style-type: none"><li>• <b>prüfe</b> ob Bezeichner <b>deklariert</b> ist</li><li>• <b>verweise</b> auf <b>bindende</b> Deklaration</li><li>• return diese <b>Deklaration</b></li></ul>
Operator	<code>visitOperator</code>	<ul style="list-style-type: none"><li>• <b>prüfe</b> ob Operator <b>deklariert</b> ist</li><li>• <b>verweise</b> auf <b>bindende</b> Deklaration</li><li>• return diese <b>Deklaration</b></li></ul>

## Ersetze in Java

```
public class SomePass implements Visitor {  
    ...  
    public Object visitXYZ(XYZ x, Object arg); ...  
}
```

## durch:

```
public class SomePass implements Visitor {  
    ...  
    public Object visit(XYZ x ,Object arg); ...  
}
```

Mißverständlich: `visit` in AST-Subklasse, `visit` in Visitor

# Standardumgebung

- Wo kommen Definitionen her z.B. von ...
  - `Integer`, `Char`, `Boolean`
  - `true`, `false`
  - `putint`, `getint`
  - `+`, `-`, `*`
- Müssen vorliegen, damit Algorithmus funktionieren kann.

➔ **Vorher** definieren (leicht gesagt ...)

Entsprechende Type-Objekte als Singletons anlegen

```
public class Type {  
    private byte kind; // INT, BOOL or ERROR  
    public static final byte  
        BOOL=0, INT=1, ERROR=-1;  
  
    private Type(byte kind) { ... }  
  
    public boolean equals(Object other) { ... }  
  
    public static Type boolT = new Type(BOOL);    // eingebaute Typen!  
    public static Type intT  = new Type(INT);  
    public static Type errorT = new Type(ERROR);  
}
```

Damit jetzt möglich

```
// Type denoter checking  
public Object visitSimpleTypeDen (SimpleTypeDen den, Object arg) {  
    if (den.l.spelling.equals("Integer")  
        den.type = Type.intT;  
    else if (den.l.spelling.equals("Boolean")  
        den.type = Type.boolT;  
    else {  
        // error: unknown type denoter  
        den.type = Type.errorT;  
    }  
    return den.type;  
}
```

...

## Handhabung von Standardumgebung

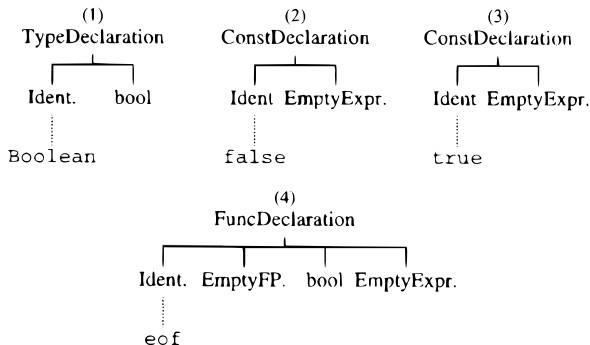
- Einlesen von Definitionen aus Quelltext
  - Ada, Haskell, VHDL, ...
- Direkt im Compiler implementiert
  - Pascal, teilweise C, Java, ...
  - (mini)-Triangle
- In beiden Fällen
  - Primitive Operationen nicht weiter in Eingabesprache beschreibbar
    - ↳ “black boxes”, nur Deklarationen sichtbar
- Geltungsbereich der Standardumgebung
  - Ebene 0: Um gesamtes Programm herum **oder**
  - Ebene 1: Auf Ebene der globalen Deklarationen im Programm



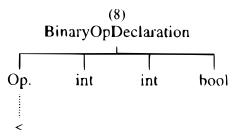
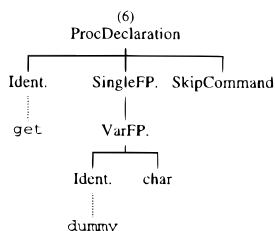
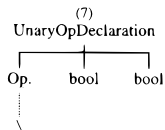
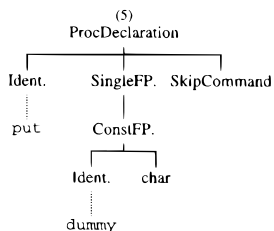
# Triangle

- Idee: Trage **Deklarationen** vorher direkt in AST ein
- Wohlgemerkt: **Ohne** konkrete Realisierung
  - Behandlung als Sonderfälle während Optimierung und Code-Erzeugung
- Deklarationen als Sub-ASTs **ohne** Definition

Beispiel: `Boolean`, `false`, `true`, `eof()` : `Boolean`



Beispiel: `put (c)`, `get (var c)`, `\ b, e1 < e2`



## Eintragen der Umgebung am Anfang der syntaktischen Analyse

```
private void establishStdEnvironment () {  
    // idTable.startIdentification();  
    StdEnvironment.booleanType = new BoolTypeDenoter(dummyPos);  
    StdEnvironment.integerType = new IntTypeDenoter(dummyPos);  
    StdEnvironment.charType = new CharTypeDenoter(dummyPos);  
    StdEnvironment.anyType = new AnyTypeDenoter(dummyPos);  
    StdEnvironment.errorType = new ErrorTypeDenoter(dummyPos);  
  
    StdEnvironment.booleanDecl = declareStdType("Boolean", StdEnvironment.booleanType);  
    StdEnvironment.falseDecl = declareStdConst("false", StdEnvironment.booleanType);  
    StdEnvironment.trueDecl = declareStdConst("true", StdEnvironment.booleanType);  
    StdEnvironment.notDecl = declareStdUnaryOp("\", StdEnvironment.booleanType, StdEnvironment.booleanType);
```

## Anlegen einer vorbelegten Konstante

```
// Creates a small AST to represent the "declaration" of a standard  
// type, and enters it in the identification table.
```

```
private ConstDeclaration declareStdConst (String id, TypeDenoter constType) {  
  
    IntegerExpression constExpr;  
    ConstDeclaration binding;  
  
    // constExpr used only as a placeholder for constType  
    constExpr = new IntegerExpression(null, dummyPos);  
    constExpr.type = constType;  
    binding = new ConstDeclaration(new Identifier(id, dummyPos), constExpr, dummyPos);  
    idTable.enter(id, binding);  
    return binding;  
}
```

## Mini-Triangle: Nur primitive Typen

- Einfach:
- Beispiel: `if E1 = E2 then ...`
- Typen von  $E1$  und  $E2$  müssen identisch sein
- `e1.type == e2.type`

Triangle ist komplizierter:  
Arrays, Records, benutzdefinierte Typen

## Beispiel 1

```
type T1 ~ record n: Integer; c: Char end;  
type T2 ~ record c: Char; n: Integer end;  
  
var t1 : T1; var t2 : T2;  
  
if t1 = t2 then ...
```

Legal?



## Beispiel 2

```
type Word ~ array 8 of Char;
```

```
var w1 : Word;
```

```
var w2 : array 8 of Char;
```

```
if w1 = w2 then ...
```

Legal?

➔ Wann sind zwei Typen äquivalent?

# 1. Möglichkeit: Strukturelle Typäquivalenz

Typen sind genau dann äquivalent, wenn ihre **Struktur** äquivalent ist.

- Primitive Typen: Müssen identisch sein
- Arrays: Äquivalenter Typ für Elemente, gleiche Anzahl
- Records: Gleiche Namen für Elemente, äquivalenter Typ für Elemente, gleiche Reihenfolge der Elemente

## 2. Möglichkeit: Typäquivalenz über Namen

Jedes Vorkommen eines nicht-primitiven Typs (selbstdefiniert, Array, Record) beschreibt einen neuen und **einzigartigen** Typ, der nur zu sich selbst äquivalent ist.

In Triangle: strukturelle Typäquivalenz

## Beispiel 1

```
type T1 ~ record n: Integer; c: Char end;  
type T2 ~ record c: Char; n: Integer end;  
  
var t1 : T1; var t2 : T2;  
  
if t1 = t2 then ...
```

Struktur nicht äquivalent, Namen nicht äquivalent

## Beispiel 2

```
type Word ~ array 8 of Char;
```

```
var w1 : Word;
```

```
var w2 : array 8 of Char;
```

```
if w1 = w2 then ...
```

Struktur äquivalent, Namen nicht äquivalent

## Beispiel 3

```
type Word ~ array 8 of Char;
```

```
var w1 : Word;
```

```
var w2 : Word;
```

```
if w1 = w2 then ...
```

Struktur äquivalent, Namen äquivalent

- Einfache Klasse **Type** reicht nicht mehr aus
- Kann beliebig kompliziert werden
- Idee: Verweis auf Typbeschreibung im AST
- Abstrakte Klasse **TypeDenoter**, Unterklassen
  - **IntegerTypeDenoter**
  - **ArrayTypeDenoter**
  - **RecordTypeDenoter**
  - ...

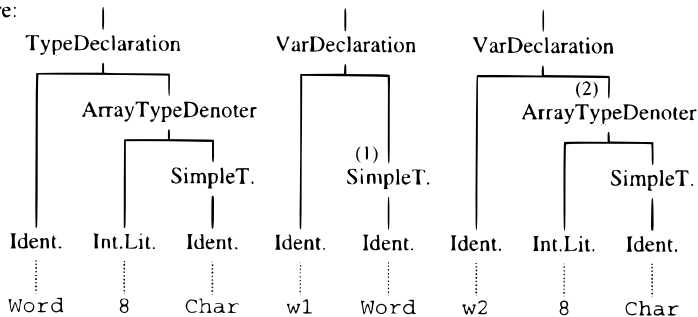
## Vorgehen

- 1 Ersetze in Kontextanalyse alle Typenbezeichner durch Verweise auf Sub-ASTs der Typdeklaration
- 2 Führe Typprüfung durch strukturellen Vergleich der Sub-ASTs der Deklarationen durch

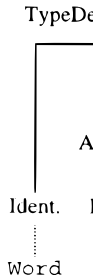


# Beispiel komplexe Typäquivalenz

Before:



(2) After:



Nun durch Vergleich während Graphdurchlauf überprüfbar.

# Zusammenfassung

- Kontextanalyse
- Identifikation
- Typüberprüfung
- Organisation von Symboltabellen
- Implementierung von AST-Durchläufen