

# Einführung in den Compilerbau

## Kontextuelle Analyse



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

WS 2018/19

Andreas Koch

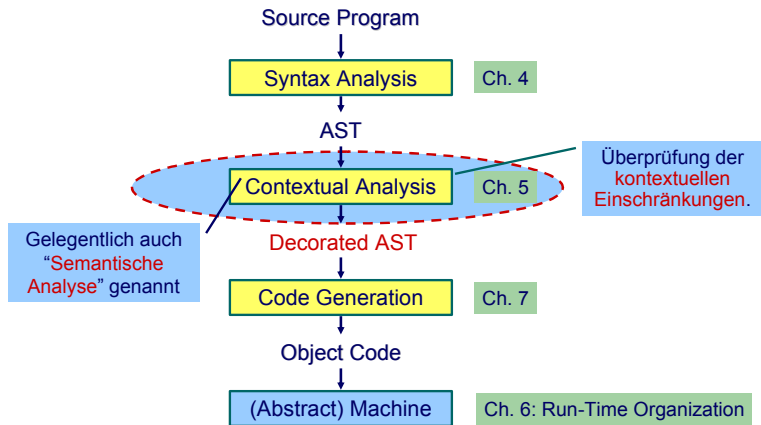
FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt



Embedded Systems & Applications



# Einleitung



# Kontextuelle Einschränkungen: Geltungsbereiche

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

Deklaration von n:  
Bindung

Benutzung von n:  
Verwendung

??

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

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

Verwendung von m



## 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

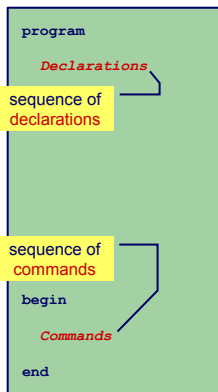




- ▶ **Geltungsbereich** von Zuordnung von Namen zu Attributen innerhalb des Programmes
- ▶ **Block** Konstrukt im Programmtext zur Beschreibung von Geltungsbereichen
  - ▶ In Triangle:  
**let** Declarations **in** Commands  
**proc** P ( formal-parameters ) ~ Commands
  - ▶ In Java:  
Geltungsbereiche 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



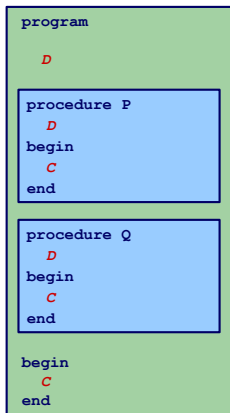
```
public class Attribute {
    // Attribute details
    ...
}

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) { ... }

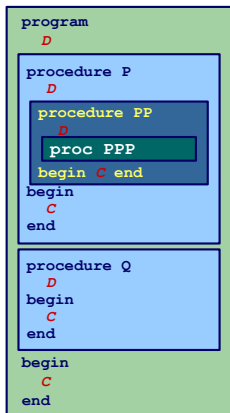
    ...
}
```



- ▶ Charakteristika
  - ▶ Mehrere überlappungsfreie Blöcke
  - ▶ Zwei Geltungsbereiche: Global und Lokal
- ▶ Regeln für Geltungsbereiche
  - ▶ Global/lokal deklarierte Bezeichner dürfen nicht global/im selben Block redefiniert werden
  - ▶ Jeder benutzte Bezeichner muss global oder lokal zu seiner Verwendungsstelle deklariert sein
- ▶ Symboltabelle
  - ▶ Bis zu zwei Einträge pro 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 Deklaration im lokalen oder **umschliessenden** Block
- ▶ Symboltabelle
  - ▶ **Mehrere** Einträge je Bezeichner möglich
  - ▶ Aber maximal ein Paar (Tiefe, 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
```

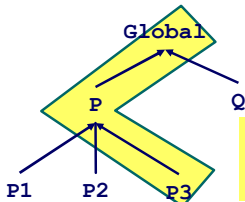
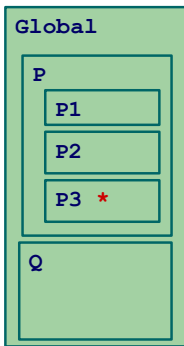
Geltungsbereiche  
und Sichtbarkeit

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
    - ▶ Ursprünglich in Triangle verwendet (natürlich ...)
  - ▶ Hier: Bessere Möglichkeiten mit Hash-Tabelle (effizienter)
  - ▶ Hash-Tabelle, die Stacks enthält
- ▶ Design-Kriterium
  - ▶ Gleiche Bezeichner tauchen häufiger in Tabelle auf
  - ▶ Aber auf unterschiedlichen Ebenen
  - ▶ Abgerufen wird immer der am tiefsten gelegene



```
public final class IdentificationTable {  
    private Map<String, Stack<Attribute>> idents;  
    private Stack<List<String>> scopes;  
    ...  
}
```

## idents

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



```
public final class IdentificationTable {  
    private Map<String, Stack<Attribute>> idents;  
    private Stack<List<String>> scopes;  
    ...  
}
```

### scopes

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



# 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
```



### 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)

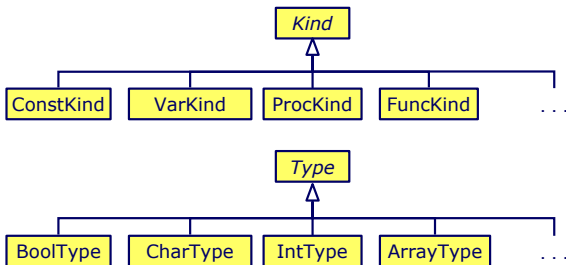
```
public class Attribute {  
  
    public static final byte // kind  
        CONST = 0,  
        VAR   = 1,  
        PROC  = 2,  
        ... ;  
  
    public static final byte // type  
        BOOL  = 0,  
        CHAR  = 1,  
        INT   = 2,  
        ARRAY = 3,  
        ... ;  
  
   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

OK für sehr einfache  
Sprachen

## Objektorientierter Ansatz (explizite Speicherung)

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

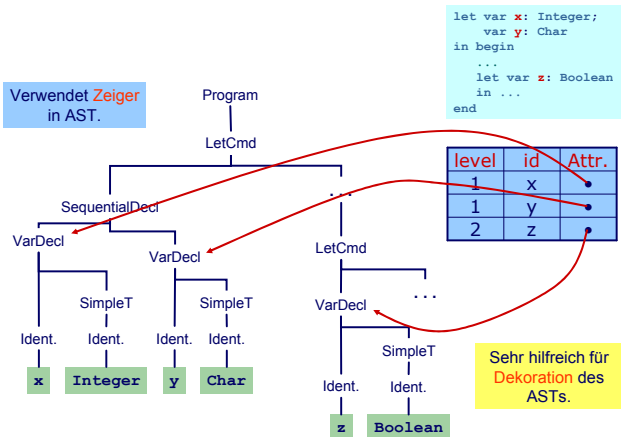
Funktioniert, wird aber bei  
realistischer Sprache  
sehr leicht unhandlich





- ▶ 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





# 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

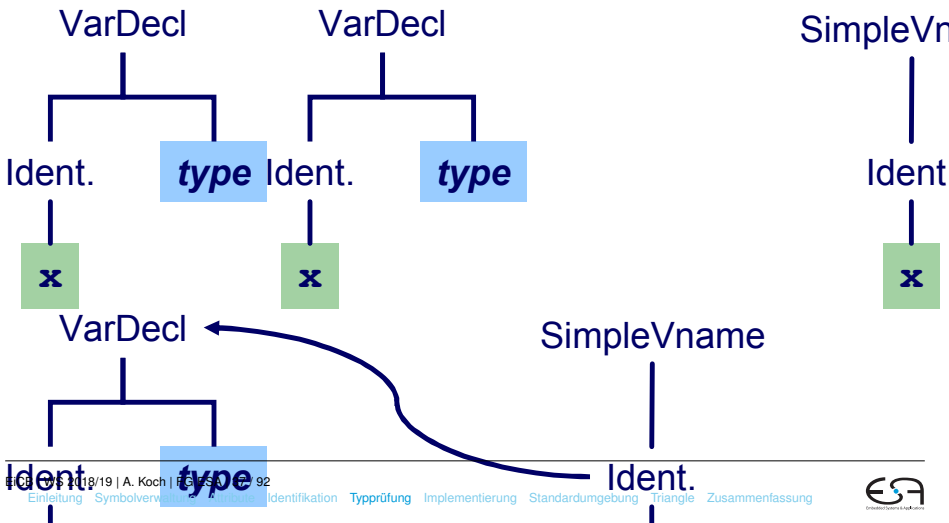
1. Berechne oder leite Typen von Ausdrücken her
  - ▶ Aus den Typen der Teilausdrücke und der Art der Verknüpfung
2. Überprüfe, das Typen der Ausdrücke Anforderungen aus dem Kontext genügen
  - ▶ Beispiel: Bedingung in **if/then** muß einen Boolean liefern



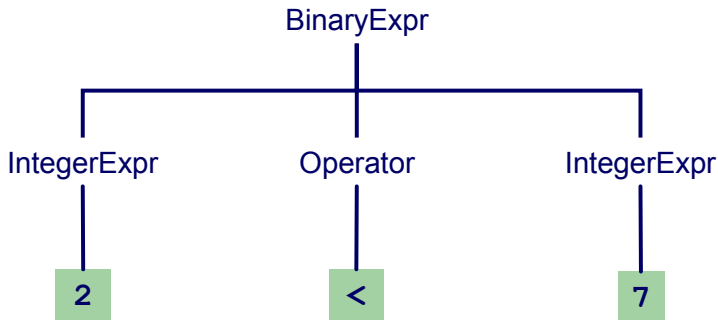
Genauer: Bottom-Up Verfahren für statisch typisierte Programmiersprache

- ▶ 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



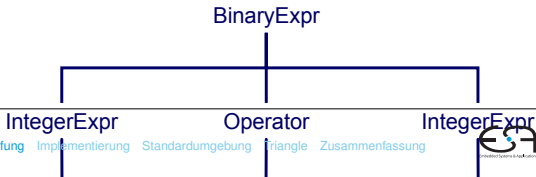
Typregel für Binären Ausdruck:

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





Anweisungen mit Ausdrücken

Typregel für **ifCommand**:

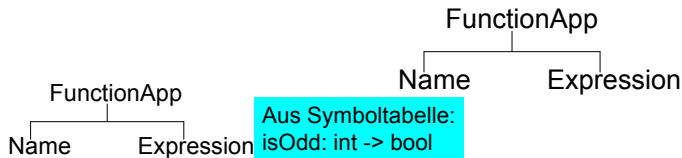
**if**  $E$  **then**  $C1$  **else**  $C2$

ist **typkorrekt** genau dann, wenn

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

# Beispiel: Typherleitung für Funktionsaufruf

## isOdd(42)

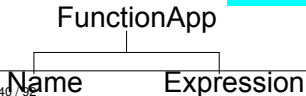


Aus Symboltabelle:  
isOdd: int -> bool

Aus Symboltabelle:  
isOdd: int -> bool

prüfe, ob vom Typ Integer

Kombiniere, dass Knoten vom Typ  
Boolean



Aus Symboltabelle:

prüfe, ob vom Typ Integer





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

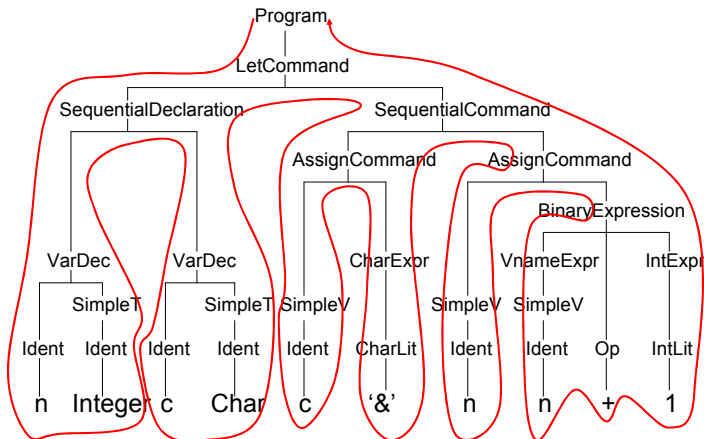
- ▶ Typprüfung des Körpers **Expression**
- ▶ Stelle sicher, dass Ergebnis von **ResultType** ist
- ▶ Dann Herleitung: **f: ParamType → ResultType**

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

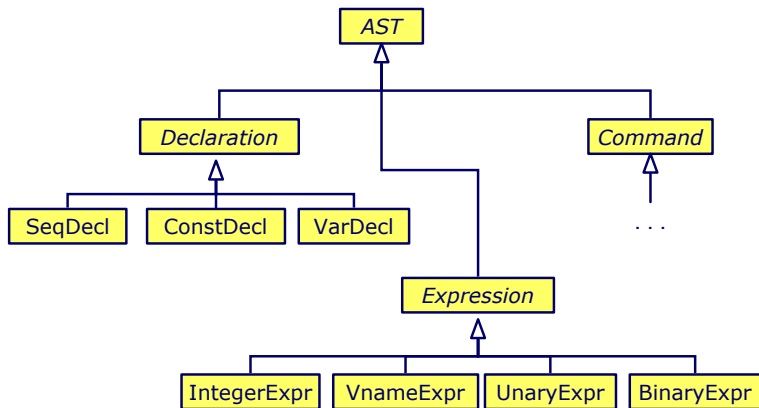




Program	::= Command	Program
Command	::= Command ; Command   V-name := Expression   Identifier ( Expression )   if Expression then single-Command   else single-Command   while Expression do single-Command   let Declaration in single-Command	SequentialCmd AssignCmd CallCmd IfCmd WhileCmd LetCmd
Expression	::= Integer-Literal   V-name   Operator Expression   Expression Operator Expression	IntegerExpr VnameExpr UnaryExpr BinaryExpr
V-name	::= Identifier	SimpleVname
Declaration	::= Declaration ; Declaration   <b>const</b> Identifier ~ Expression   <b>var</b> Identifier : Type-denoter	SeqDecl ConstDecl VarDecl
Type-denoter	::= Identifier	SimpleTypeDen

Grammatik von  
abstrakter Syntax  
von Mini-Triangle

AST Knoten von Mini-Triangle





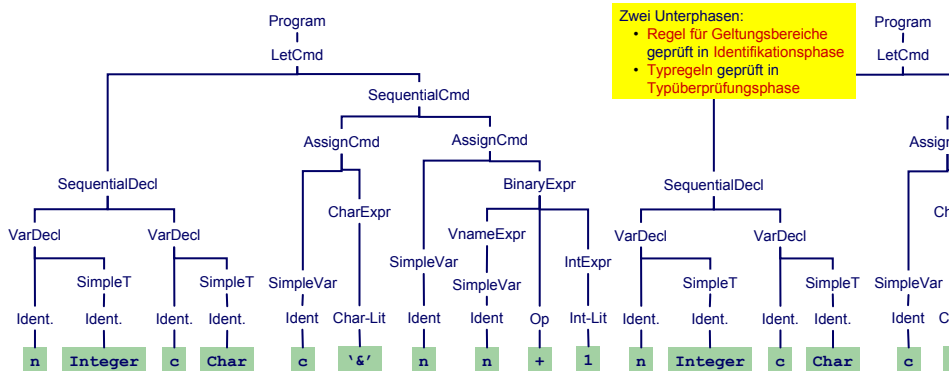
Expression ::= Integer-Literal	IntegerExpr
V-name	VnameExpr
Operator Expression	UnaryExpr
Expression Operator Expression	BinaryExpr

```
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 zusätzliche 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);
```

Rückgabe

au

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

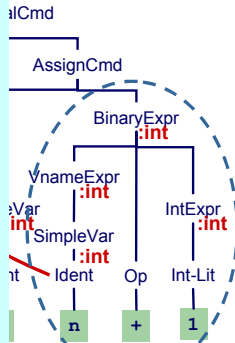
# Beispiel: Dekorierung via OO Ansatz



```
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[0]  
Type

## 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 class AST {  
    public abstract <RetTy, ArgTy> RetTy visit(Visitor<RetTy, ArgTy> v,  
        ArgTy arg);  
}  
public class AssignCommand extends Command {  
    public <RetTy, ArgTy> RetTy visit(Visitor<RetTy, ArgTy> v, ArgTy arg) {  
        return v.visitAssignCommand(this, arg);  
    }  
}
```

Unterschiedliche Implementierungen der Methode realisieren die geforderte Funktionalität (Typüberprüfung, Code-Erzeugung, ...)

```
public interface Visitor<RetTy, ArgTy> {  
    RetTy visitProgram          (Program prog,          ArgTy arg);  
    RetTy visitAssignCommand    (AssignCommand cmd,    ArgTy arg);  
    RetTy visitSequentialCommand(SequentialCommand cmd, ArgTy arg);  
    ...  
    RetTy visitVnameExpression  (VnameExpression expr, ArgTy arg);  
    RetTy visitBinaryExpression (BinaryExpression expr, ArgTy arg);  
    ...  
}
```

- ▶ Allgemeines Schema: Visitor-Interface definiert visitXYZ für alle Subklassen XYZ von AST **public** RetTy visitXYZ(XYZ x, ArgTy arg);
- ▶ visitXYZ wird von visit-Methode aufgerufen, die jede Klasse XYZ überschreibt:

```
public class XYZ extends ... {  
    public <R, A> R visit(Visitor<R, A> v, A arg) {  
        return v.visitXYZ(this, arg);  
    }  
}
```





- ▶ Erster Ansatz:

```
public class Checker implements Visitor<AST, AST> {  
    private IdentificationTable idTab;  
  
    public void check(Program prog) {  
        idTab = new IdentificationTable();  
        prog.visit(this, null);  
    }  
  
    ... // Implementierung der Visitor-Methoden  
}
```

- ▶ Problem (vorweg): Im AST werden unterschiedliche Informationen durch die Rückgabewerte und Argumente propagiert.
- ▶ Durch AST als Typparameter hat man nicht viel gewonnen.

# Kontextanalyse mit mehr Typsicherheit (bei der Implementierung)



- ▶ Einsicht: Für `Command`, `Expression` und die anderen abstrakten Unterklassen von `AST` kann man jeweils spezifischere Rückgabe- und Argumenttypen finden
- ▶ Also: Führe spezialisierte Visitorimplementierungen ein, die nur einen Teil der `AST`-Knoten behandelt
  - ▶ Definiere dazu abstrakte Klasse `VisitorBase`, die alle Methoden des Interfaces durch Werfen einer Exception implementiert.

# Kontextanalyse mit mehr Typsicherheit (bei der Implementierung)



```
public class Checker {
    private IdentificationTable idTab;
    public void check(Program prog) {
        idTab = new IdentificationTable();
        prog.visit(programChecker, null);
    }

    private class CommandChecker extends VisitorBase<Void, Void> {
        public Void visitAssignCommand(AssignCommand cmd, Void __) { ... }
        ...
    }
    private class ExpressionChecker extends VisitorBase<TypeDenoter,
        Void> { ... }

    private CommandChecker    commandChecker = new CommandChecker();
    private ExpressionChecker exprChecker    = new ExpressionChecker();
    ...
    private ProgramChecker    programChecker = new ProgramChecker();
}
```

# Beispiel: AssignCommand



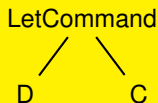
AssignCommand



```
private class CommandChecker extends VisitorBase<Void, Void> {  
    public Void visitAssignCommand(AssignCommand ast, Void __) {  
        TypeDenoter vType = ast.V.visit(vnameChecker, null);  
        TypeDenoter eType = ast.E.visit(expressionChecker, null);  
        if (! ast.V.variable)  
            reporter.reportError("LHS of assignment is not a  
variable", "", ast.V.position);  
        if (! eType.equals(vType))  
            reporter.reportError("assignment incompatibility", "",  
ast.position);  
        return null;  
    }  
}
```

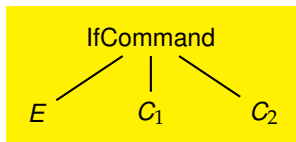
...

# Beispiel: LetCommand



```
private class CommandChecker extends VisitorBase<Void, Void> {  
    ...  
    public Void visitLetCommand(LetCommand ast, Void __) {  
        idTable.openScope();  
        ast.D.visit(declarationChecker, null);  
        ast.C.visit(commandChecker, null);  
        idTable.closeScope();  
        return null;  
    }  
    ...  
}
```

LetCommand **öffnet** (und **schließt**) eine Ebene von Geltungsbereichen in **Symboltabelle**



```
private class CommandChecker extends VisitorBase<Void, Void> {  
    ...  
    public Void visitIfCommand(IfCommand ast, Void __) {  
        TypeDenoter eType = ast.E.visit(expressionChecker, null);  
        if (! eType.equals(StdEnvironment.booleanType))  
            reporter.reportError("Boolean expression expected here",  
                "", ast.E.position);  
        ast.C1.visit(commandChecker, null);  
        ast.C2.visit(commandChecker, null);  
        return null;  
    }  
    ...  
}
```

# Beispiel: IntegerExpression

IntegerExpression

|  
IL

```
private class ExpressionChecker extends  
    VisitorBase<TypeDenoter, Void> {  
    ...  
    public TypeDenoter visitIntegerExpression(IntegerExpression  
        ast, Void __) {  
        ast.type ← ast.IL.visit(literalChecker, null);  
        return ast.type;  
    }  
    ...
```

Dekoriere den IntegerExpression-Knoten im AST

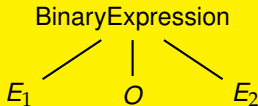
# Beispiel: BinaryExpression



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
private class ExpressionChecker extends VisitorBase<TypeDenoter, Void> {
    ...
    public TypeDenoter visitBinaryExpression(BinaryExpression ast, Void ...) {
        TypeDenoter e1Type = ast.E1.visit(expressionChecker, null);
        TypeDenoter e2Type = ast.E2.visit(expressionChecker, null);
        Declaration binding = ast.O.visit(identifierOperatorChecker, null);

        if (binding == null)
            reportUndeclared(ast.O);
        else {
            if (!(binding instanceof BinaryOperatorDeclaration))
                reporter.reportError("\"%\" is not a binary operator", ast.O.spelling, ast.O.position);
            BinaryOperatorDeclaration bbinding = (BinaryOperatorDeclaration) binding;
            if (bbinding.ARG1 == StdEnvironment.anyType) {
                // this operator must be "=" or "\="
                if (! e1Type.equals(e2Type))
                    reporter.reportError("incompatible argument types for \"%\"", ast.O.spelling, ast.position);
            }
            else if (! e1Type.equals(bbinding.ARG1))
                reporter.reportError("wrong argument type for \"%\"", ast.O.spelling, ast.E1.position);
            else if (! e2Type.equals(bbinding.ARG2))
                reporter.reportError("wrong argument type for \"%\"", ast.O.spelling, ast.E2.position);
            ast.type = bbinding.RES;
        }
        return ast.type;
    }
    ...
}
```



Weitere Beispiele siehe Triangle-Compiler-Code



# Beispiel: VarDeclaration und ConstDeclaration

```
private class DeclarationChecker extends VisitorBase<Void, Void> {
    public Void visitConstDeclaration(ConstDeclaration ast, Void __) {
        ast.E.visit(expressionChecker, null);
        boolean duplicated = idTable.enter(ast.I.spelling, ast);
        if (duplicated)
            reporter.reportError("identifier \"%\" already declared",
                ast.I.spelling, ast.position);
        return null;
    }

    public Void visitVarDeclaration(VarDeclaration ast, Void __) {
        ast.T = ast.T.visit(typeDenoterChecker, null);
        boolean duplicated = idTable.enter(ast.I.spelling, ast);
        if (duplicated)
            reporter.reportError("identifier \"%\" already declared",
                ast.I.spelling, ast.position);

        return null;
    }
}
```

## Beispiel: SimpleVname



```
private class VnameChecker extends VisitorBase<TypeDenoter, Void> {
    public TypeDenoter visitSimpleVname(SimpleVname ast, Void __) {
        ast.variable = false;
        ast.type = StdEnvironment.errorType;
        Declaration binding = ast.I.visit(identifierOperatorChecker, null);
        if (binding == null)
            reportUndeclared(ast.I);
        else if (binding instanceof EntityDeclaration) {
            EntityDeclaration entDecl = (EntityDeclaration) binding;
            ast.type = entDecl.getType();
            ast.variable = ! entDecl.isConstant();
        }
        else
            reporter.reportError("\'%\' is not a const or var identifier",
                ast.I.spelling, ast.I.position);
        return ast.type;
    }
    ...
}
```

- ▶ EntityDeclaration ist ein Interface, das u.a. von VarDeclaration und ConstDeclaration implementiert wird.

# 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>
Identifier	<code>visitIdentifier</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 <code>Deklaration</code></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 <code>Deklaration</code></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

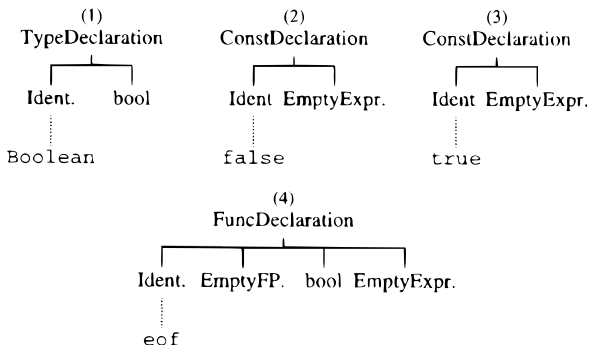


# Triangle



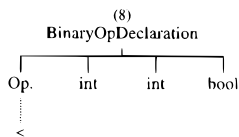
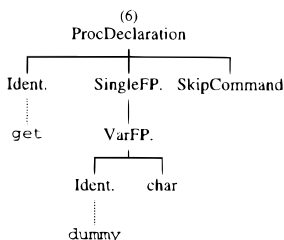
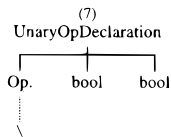
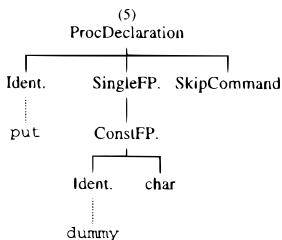
- ▶ Idee: Trage **Deklarationen** vorher direkt in AST ein
- ▶ Wohlgemerkt: **Ohne** konkrete Realisierung
  - ▶ Behandlung als Sonderfälle während Optimierung und Code-Erzeugung
- ▶ Deklarationen als Sub-ASTs **ohne** Definition

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



# Standardumgebung: Realisierung in Triangle 3

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**
  - ▶ ...



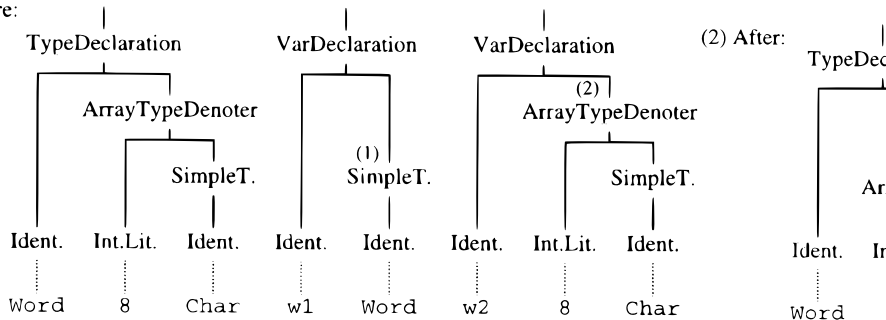


## Vorgehen

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:



Nun durch Vergleich während Graphdurchlauf überprüfbar.



# Zusammenfassung



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