Optimierende Compiler

3. Kontextanalyse

Andreas Koch

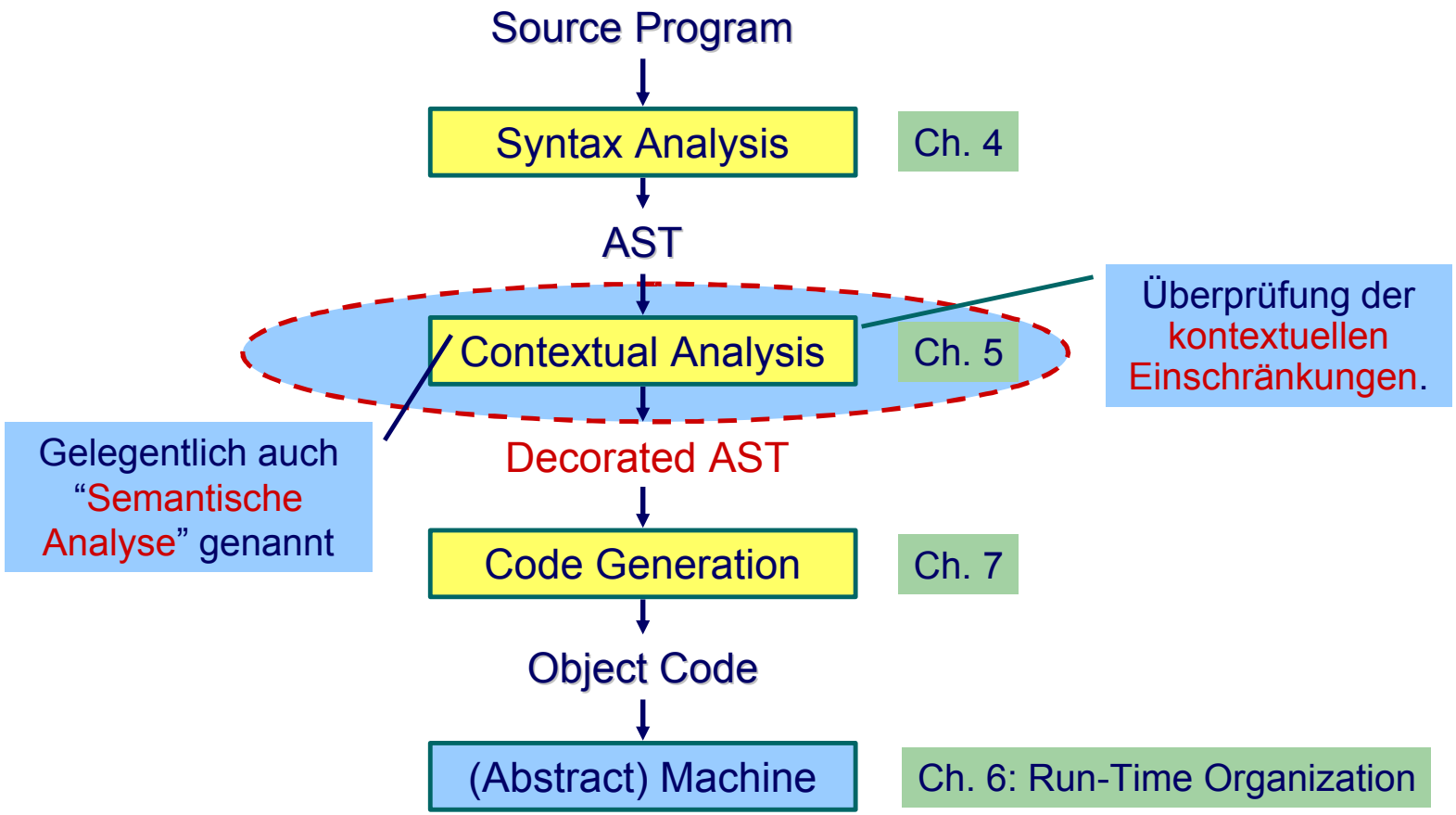
FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2006

Organisatorisches

- Abgabe 1. Aufgabe: 3.5.2006

Übersicht



Kontextuelle Einschränkungen: Geltungsbereiche

Syntaktische Korrektheit reicht nicht aus für sinnvolle Übersetzung

Geltungsbereiche (Scope)

- Betreffen *Sichtbarkeit* von Bezeichern
- Jeder verwendete Bezeichner muss vorher *deklariert* werden
  - . . . nicht bei allen Programmiersprachen
- Deklaration ist sog. *bindendes Auftreten* des Bezeichners
- Benutzung ist sog. *verwendendes Auftreten* des Bezeichners
- Aufgabe: Bringe jede Verwendung mit genau der einen passenden Bindung in Zusammenhang

Beispiele 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!

Kontexttuelle Einschränkungen: Typen

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)

Beispiele Typen

```
let
   var n: Integer
in begin
   ...
   while n > 0 do
      n := n-1;
   ...
end
```

Typregel für $E_1 > E_2$ (GreaterOp):
Wenn E_1 und E_2 beide vom Typ int, dann ist Ergebnis vom Typ bool.

Typregel für **while** E **do** C (WhileCmd):
E muss vom Typ boolean sein.

Typregel für V:=E (AssignCmd):
Die Typen von V und E müssen äquivalent sein.

Typregel für $E_1 - E_2$ (SubOp):
Wenn E_1 und E_2 beide vom Typ int, dann ist Ergebnis vom Typ int.

Was prüfen?

- 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
- …

Zuordnung von Namen zu Attributen 1

- 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, . . .

Zuordnung von Namen zu Attributen 2

- Beispiel für zugeordnete Attribute

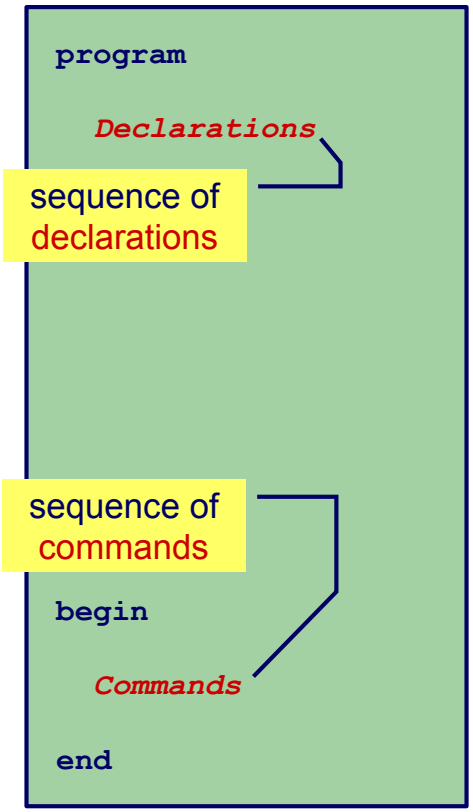
Typ	int, char, boolean, record, array pointer, ...
Art	Konstante, Variable, Funktion, Prozedur, Wert-Parameter, ...
Sichtbarkeit	Public, private, protected
Anderes	synchronized, static, volatile, ...

- Typische Operationen
- Eintragen einer neuen Zuordnung Namen-Attribute
- Abrufen der Attribute zu einem Namen
- Hierarchische Blockorganisation

Zuordnung von Namen zu Attributen 2

- 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

Monolithische Blockstruktur



- 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

Beispiel-Code Symboltabelle

```
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) { ... }

...
```

Flache Blockstruktur

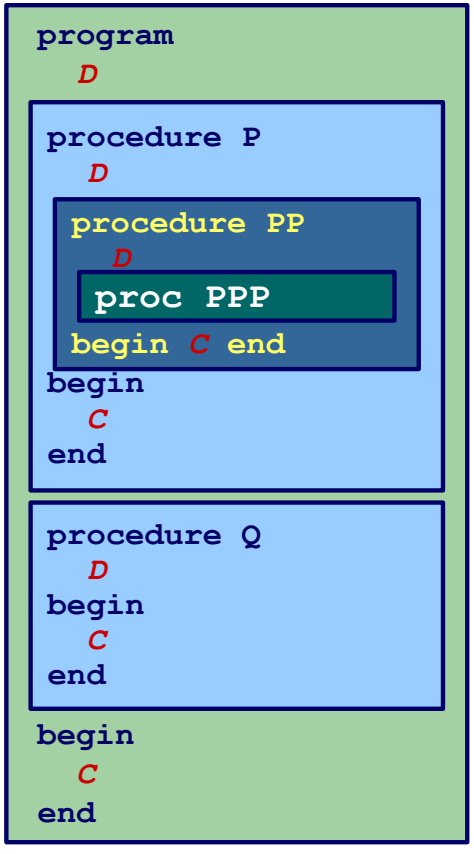
```
program
  D
  procedure P
    D
  begin
    C
  end
  procedure Q
    D
  begin
    C
  end
  begin
    C
  end
```

- Charakteristika
  - Mehrere überlappungsfreie Blöcke
  - Zwei Geltungsbereiche: Global und Lokal
- Regeln für Geltungsbereiche
  - Global deklarierte Bezeichner dürfen nicht global redeklariert werden
  - Lokal deklarierte Bezeichner dürfen nicht im selben Block redeklariert 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

Beispiel-Code Symboltabelle

```
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() { ... }
  ...
```

Verschachtelte Blockstruktur



- Charakteristika
  - Blöcke können ineinander verschachtelt sein
  - 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

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
```

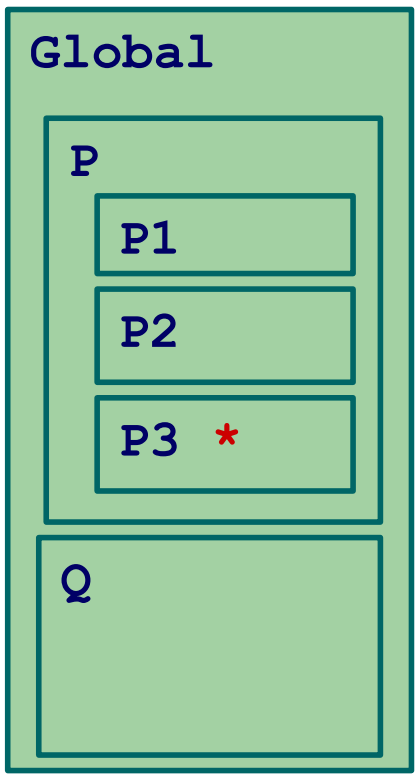
Geltungsbereiche und Sichtbarkeit

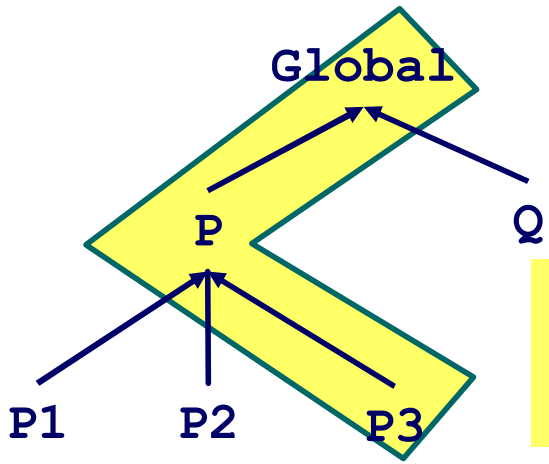
```
let !level 1
    var a, b, c ;
in begin
   let !level 2
```

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.

Beispiel-Code Symboltabelle

```
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() { ... }
  ...
```

Beispiel verschachtelte Geltungsbereiche

```
let !level 1
(1) var a: Integer;
(2) var b: Boolean
in begin
   ...
   let !level 2
   (3) var b: Integer;
   (4) var c: Boolean;
   in begin
      let !level 3
      (5) const x ~ 3
      in ...
   end
   let !level 2
   (6) var d: Boolean
   (7) var e: Integer
   in begin
end
```

Implementierung der Symboltabelle

- Verschiedene Varianten
  - Verkettete Liste und lineare Suche (einfach aber langsam)
    - Triangle (natürlich ...)
  - Hash-Tabelle (effizienter)
  - Stack aus Hash-Tabellen
- Design-Kriterium
  - Gleicher Bezeichner taucht häufiger in Tabelle auf
  - Aber auf unterschiedlichen Ebenen
  - Abgerufen wird immer der am tiefsten gelegene

Effizientere Implementierung 1

```
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;
```

Effizientere Implementierung 2

`symtab`

- 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

Effizientere Implementierung 3

`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 `symtab` (entferne jeweils oberstes Stapelelement)
  - 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

Beispiele: Verwendung von Attributen 1

Beispiel 1:

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

Beispiel 2:

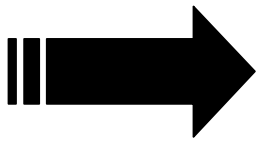
```
let const m~2 ;
    var   n:Boolean
in begin
   n := m<4;
   n := n+1
end
```

Beispiele: Verwendung von Attributen 2

Beispiel 1:

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

Undefiniert!

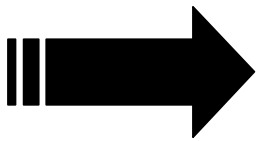


Geltungsbereichs-regeln

Beispiel 2:

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

Typfehler!



Typregeln

Beispiel: Speicherung von Attributen 1

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;
}
```

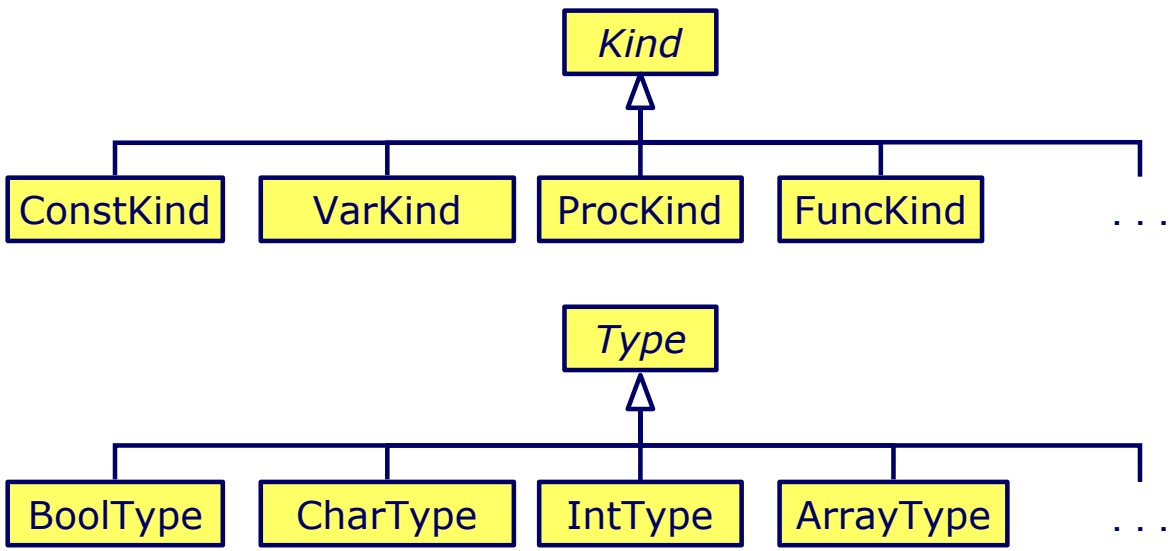
OK für sehr einfache Sprachen

Beispiel: Speicherung von Attributen 2

Objektorientierter Ansatz (explizite Speicherung)

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

Funktioniert, wird aber bei realistischer Sprache sehr leicht unhandlich



Beobachtungen

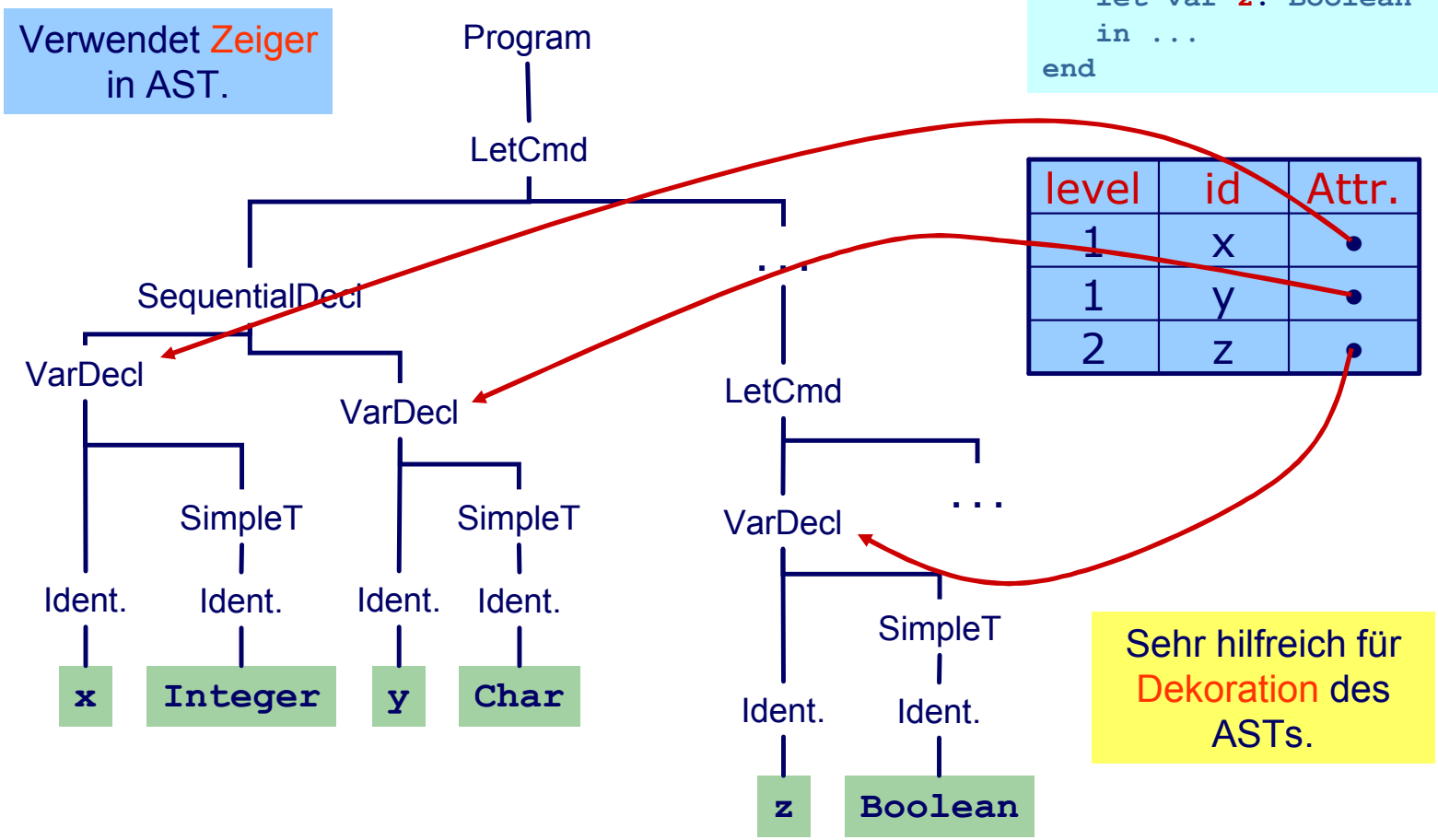
- Schon bloße Aufzählung in Form von Klassen langatmig
- Noch nicht berücksichtigt: Kombinationen
  - array [1:10] of record int x; char y end;
- Explizite Strukturen können leicht sehr komplex werden

- Idee: Im AST stehen bereits alle Daten
  - Deklarations-Unterbaum
- Als Attribute einfach Verweise auf ursprüngliche Definition eintragen
  - Dabei Geltungsbereiche beachten

AST-basierte Attribute


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

Verwendet Zeiger in AST.



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

Typen

- Was ist ein Typ?
  - “Eine Einschränkung der möglichen Interpretationen eines Speichebereiches 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, . . .

Typüberprüfung 1

- 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)

Typüberprüfung 2

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

Typüberprüfung 3

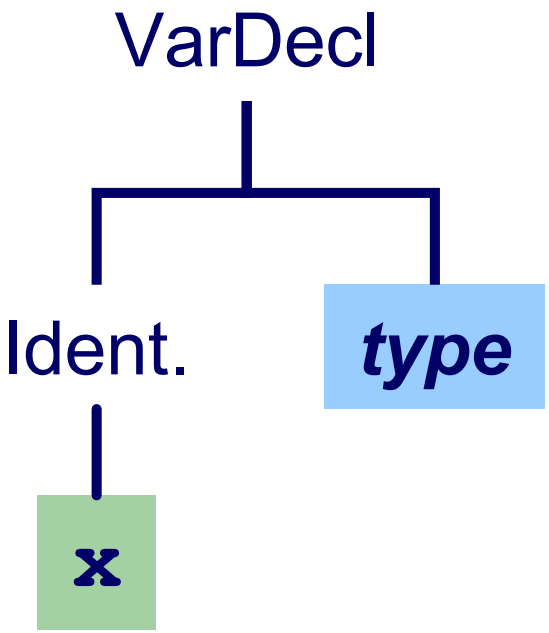
Genauer: Bottom-Up Verfahren für statisch typisierte Programmiersprache

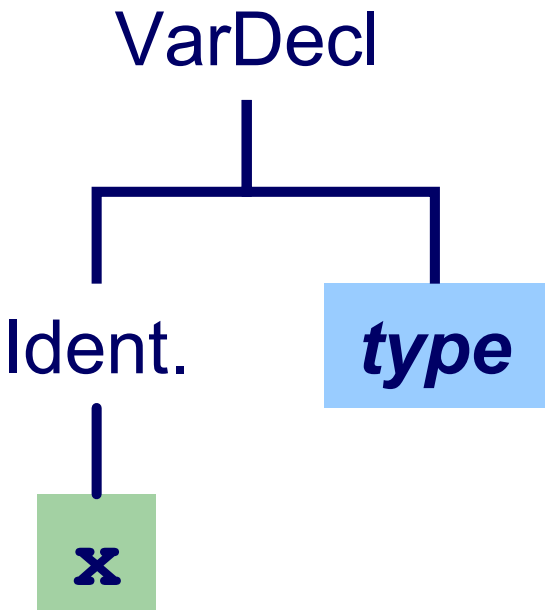
- Typen an den Blättern des AST sind bekannt

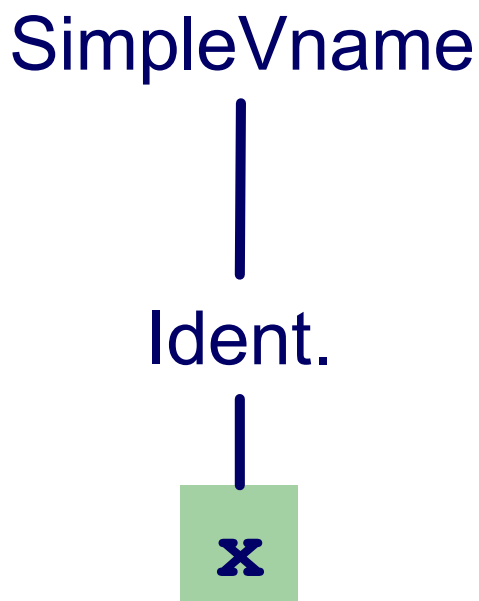
Literale	Direkt aus Knoten (true/false, 23, 42, 'a')
Variablen	Aus Symboltabelle
Konstanten	Aus Symboltabelle

- Typen der internen Knoten herleitbar aus
  - Typen der Kinder
  - Typregel für die Art der Verknüpfung im Ausdruck

Beispiel: Typherleitung für Ausdrücke







Beispiel: Typherleitung für Anweisungen

Anweisungen mit Ausdrücken

Typregel für **ifCommand**:

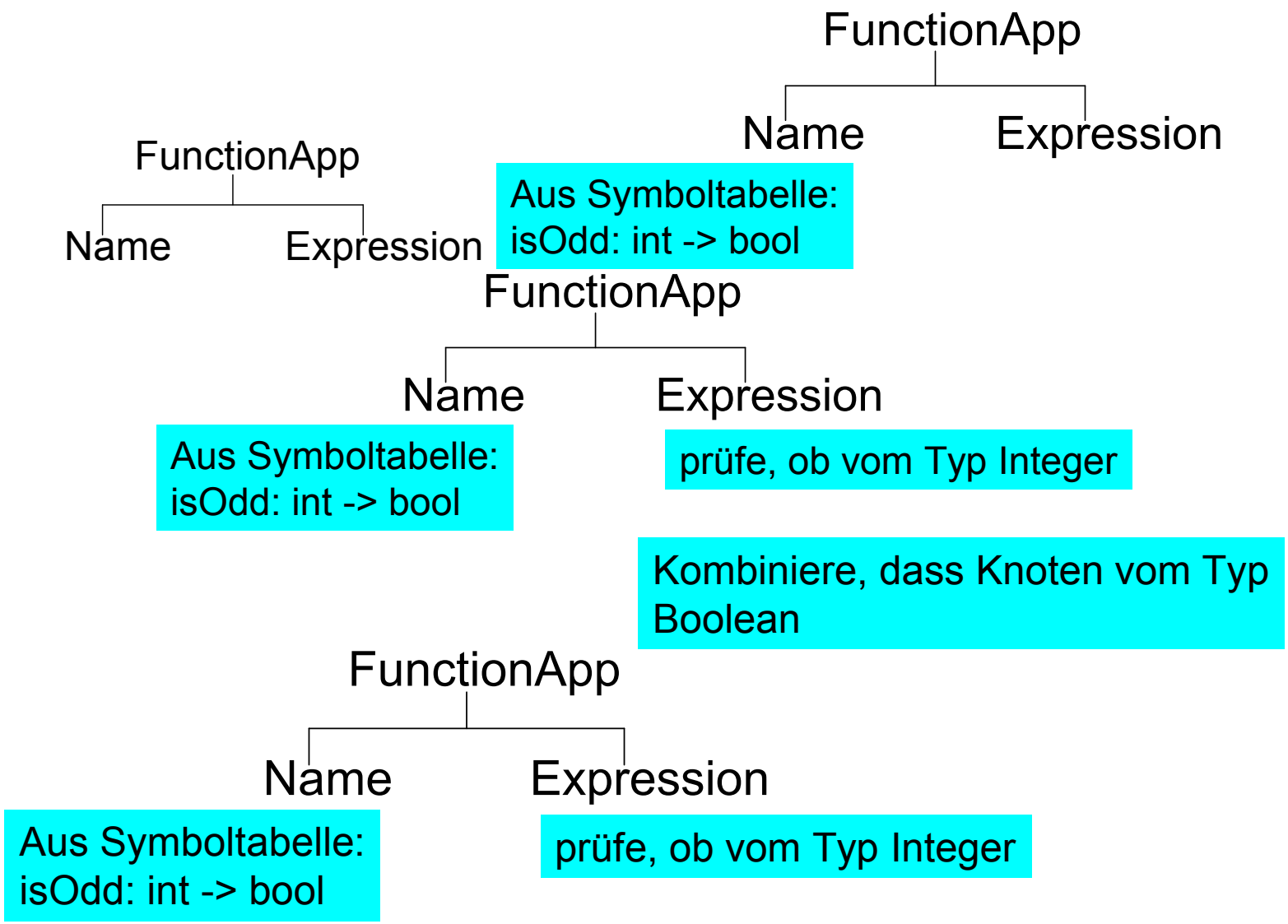
`if` *E* `then` *C1* `else` *C2*

ist typkorrekt genau dann, wenn

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

Beispiel: Typherleitung für Funktionsaufruf

`isOdd(42)`



Typüberprüfung einer Funktionsdefinition

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

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

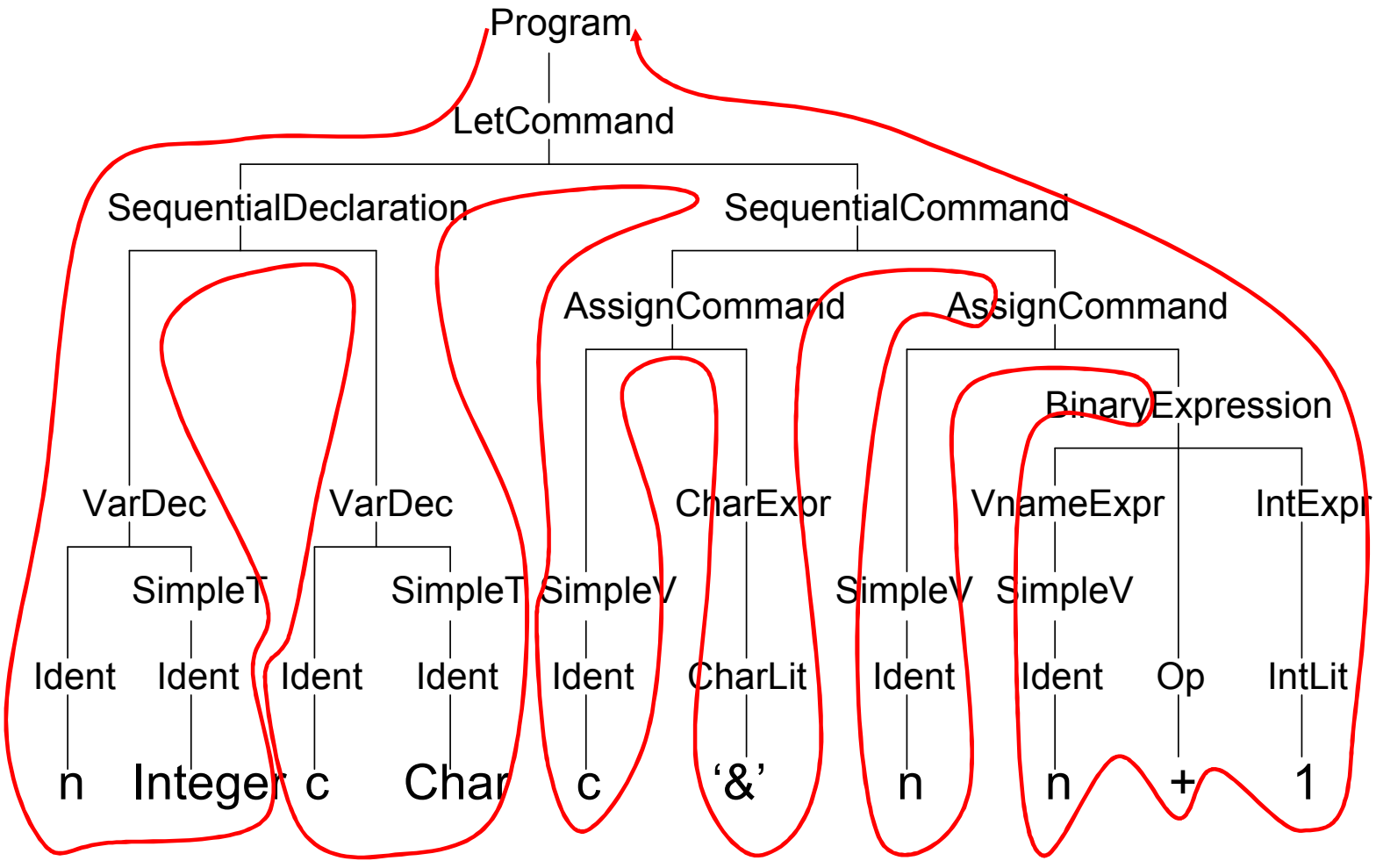
Idee: Vereinheitliche Typüberprüfung von Funktionen und Operatoren

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

Algorithmus für Kontextanalyse

- 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

AST-Durchlauf



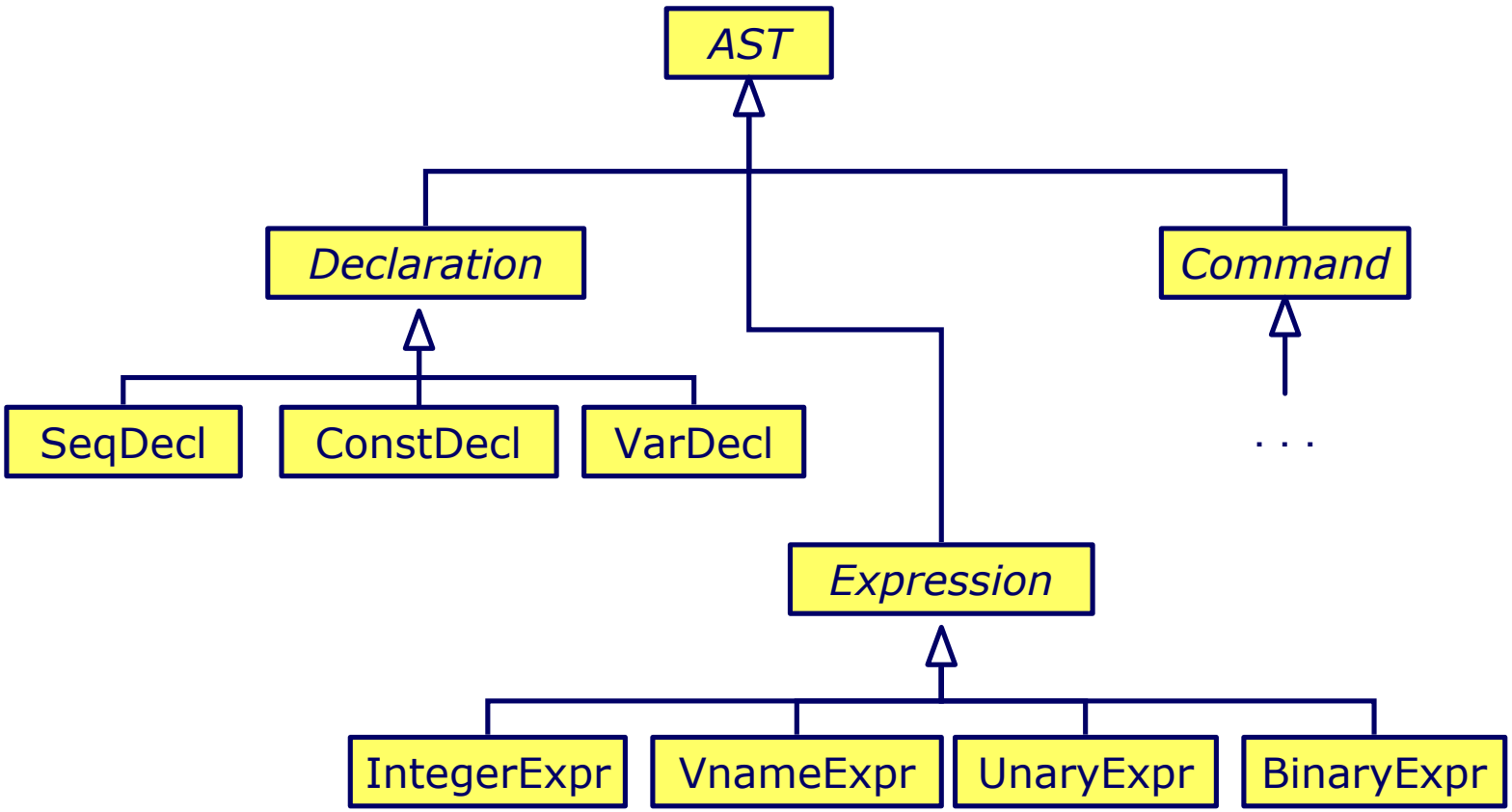
Abstrakter Syntaxbaum

Grammatik von abstrakter Syntax von Mini-Triangle

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

AST Knoten von Mini-Triangle

Klassenstruktur für AST

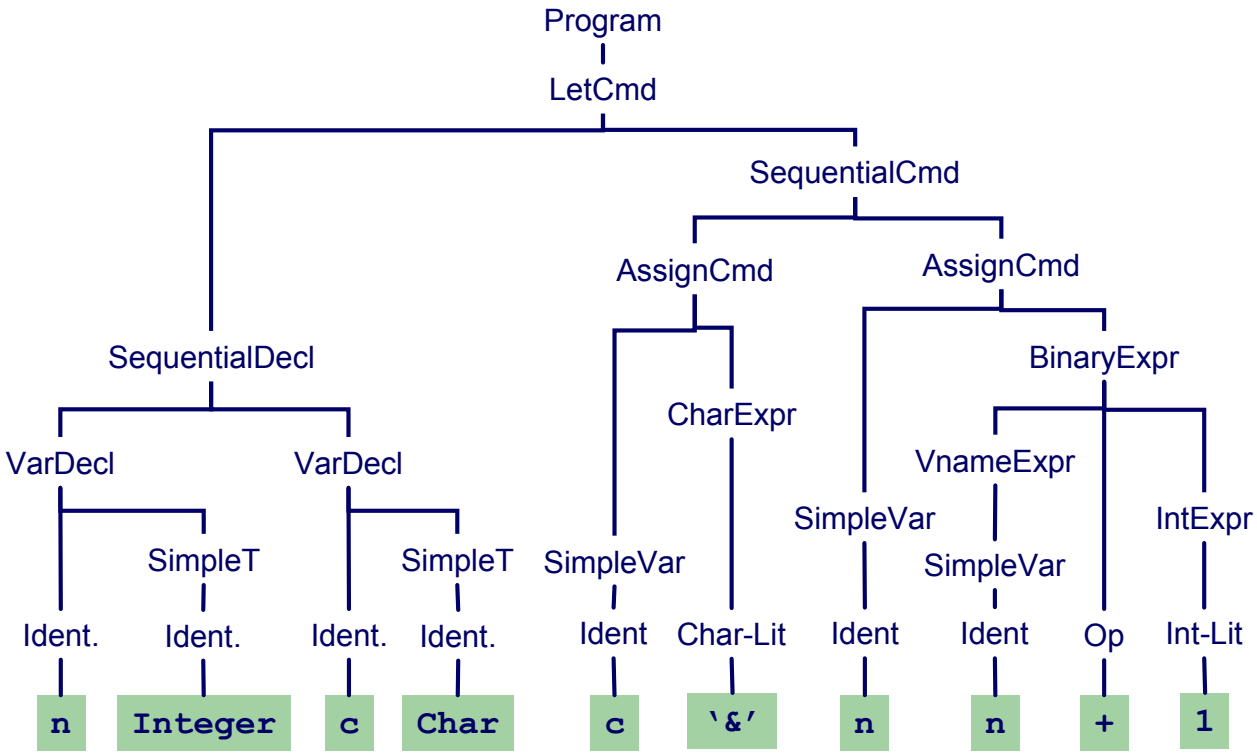


Klassendefinitionen für AST

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

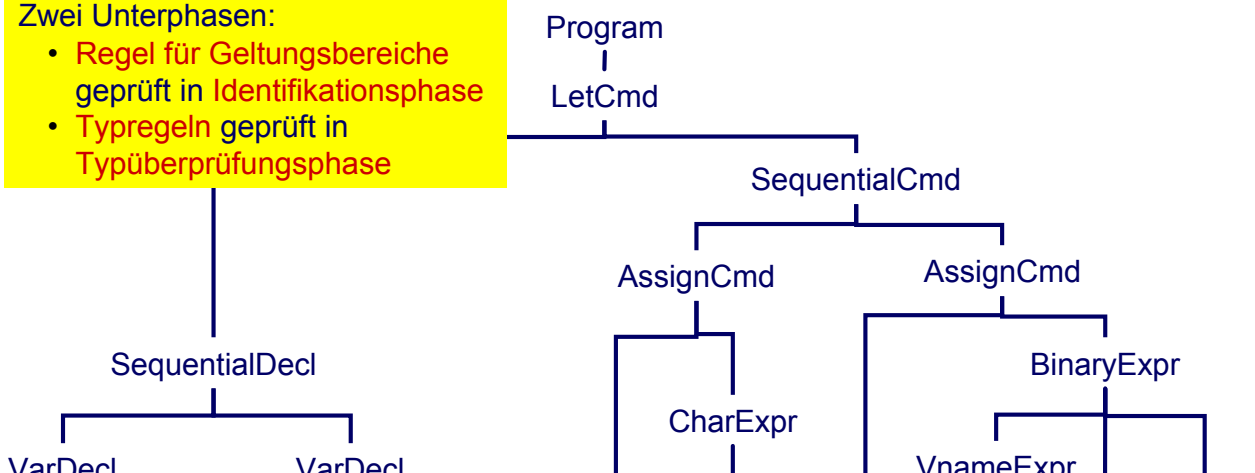
```
public class BinaryExpr extends Expression {
    public Expression E1, E2;
    public Operator   O;
}
public class UnaryExpr extends Expression {
    public Expression E;
    public Operator   O;
}
...
```

Gewünschtes Ergebnis



Zwei Unterphasen:

- Regel für Geltungsbereiche geprüft in Identifikationsphase
- Typregeln geprüft in Typüberprüfungsphase



Dekorierung des AST: Datenstruktur

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?

Dekorierung des AST: 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);
```

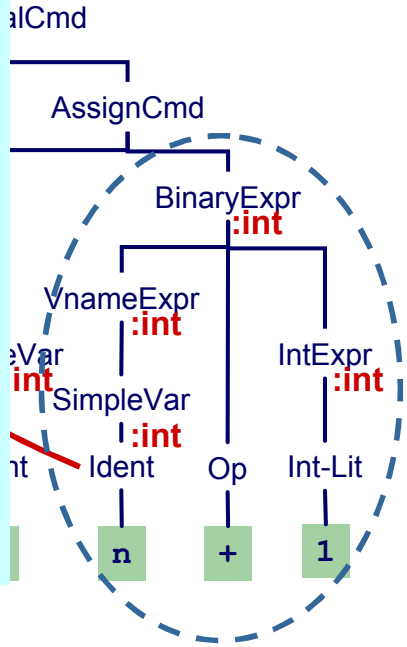
Rückgabewert propagiert Daten aufwärts im AST

Extra `arg` propagiert Daten abwärts im AST

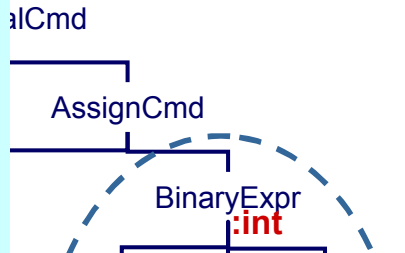
- Vorteil OO-Vorgehen leicht verständlich und

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);
```



Alternative: “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

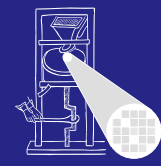
Alternative: Entwurfsmuster “Besucher”

- 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

Eigenschaften des Visitor-Pattern

- 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

Benutzung von Visitors 1



- Definiere `Visitor`-Klasse um AST-Knoten zu besuchen (`visit`)
- Füge zu jeder AST-Subklasse `XYZ` eine einzelne `visitXYZ`-Methode hinzu
  - In der Literatur auch `accept` genannt, hier Konflikt mit Parser

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

Benutzung von Visitors 2

Schema: Erweitere jede AST-Klasse `XYZ` um `visit`-Methode

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

➥Rufe `visitXYZ`-Methode des Visitors auf

Benutzung von Visitors 3

```
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);
```

Kontextanalyse als Visitor

Jetzt alle benötigten Methoden zusammen in einer Klasse

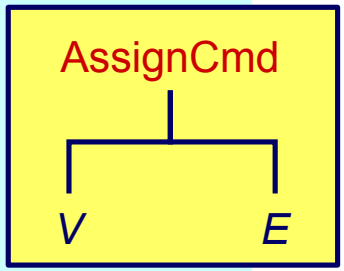
```
public class Checker implements Visitor {

    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);
  }
}
```

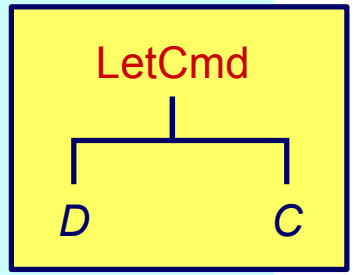
```
public Object visitAssignCmd
          (AssignCmd com, Object arg) {
   Type vType = (Type) com.V.visit(this, null);
   Type eType = (Type) com.E.visit(this, null);
   if (! com.V.variable)
       error: left side is not a variable
   if (! eType.equals(vType))
       error: types are not equivalent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

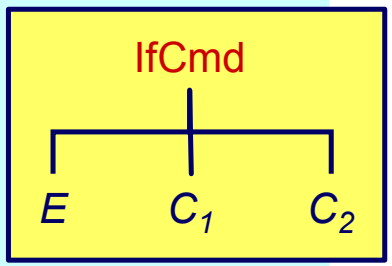


letCmd öffnet (und schließt) eine Ebene von Geltungsbereichen in Symboltabelle.

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;
}
```

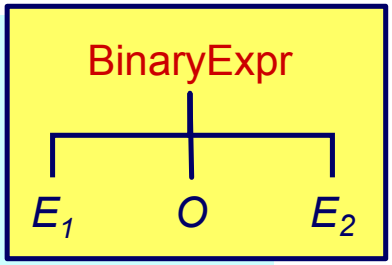
Dekoriere den IntegerExpr Knoten im AST

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.I.spelling,decl);
   return null;
}

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

...
```

Beispiel **simpleVName**

```
// VName checking
public Object visitSimpleVName (SimpleVname vname, Object arg) {
   Declaration decl = vname.I.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	**`visitProgram`**	• return **`null`**
Command	**`visit..Cmd`**	• return **`null`**
Expression	**`visit..Expr`**	• dekoriere ihn mit seinem Typ • return Typ
Vname	**`visitSimpleVname`**	• dekoriere ihn mit seinem Typ • setze Flag, falls Variable • return Typ
Declaration	**`visit..Decl`**	• trage alle deklarierten Bezeichner in Symboltabelle ein • return **`null`**
TypeDenoter	**`visit..TypeDenoter`**	• dekoriere ihn mit seinem Typ • return Typ
Identifier	**`visitIdentifier`**	• prüfe ob Bezeichner deklariert ist • verweise auf bindende Deklaration • return diese Deklaration
Operator	**`visitOperator`**	• prüfe ob Operator deklariert ist • verweise auf bindende Deklaration • return diese Deklaration

Ausnutzung von Overloading

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); ...
}
```

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

Vordefinierte Bezeichner

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

➥Vorher definieren (leicht gesagt ...)

Mini-Triangle: Eingebaute (primitive) Typen 1

Entsprechende Type-Objekte als Singletons anlegen

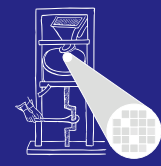
```
public class Type {
  private byte kind; // INT, BOOL or ERROR
  public static final byte
    BOOL=0, INT=1, ERROR=-1;

  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);
}
```

Mini-Triangle: Eingebaute (primitive) Typen 2

Damit jetzt möglich

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

...
```

Standardumgebung



Handhabung von Standardumgebung

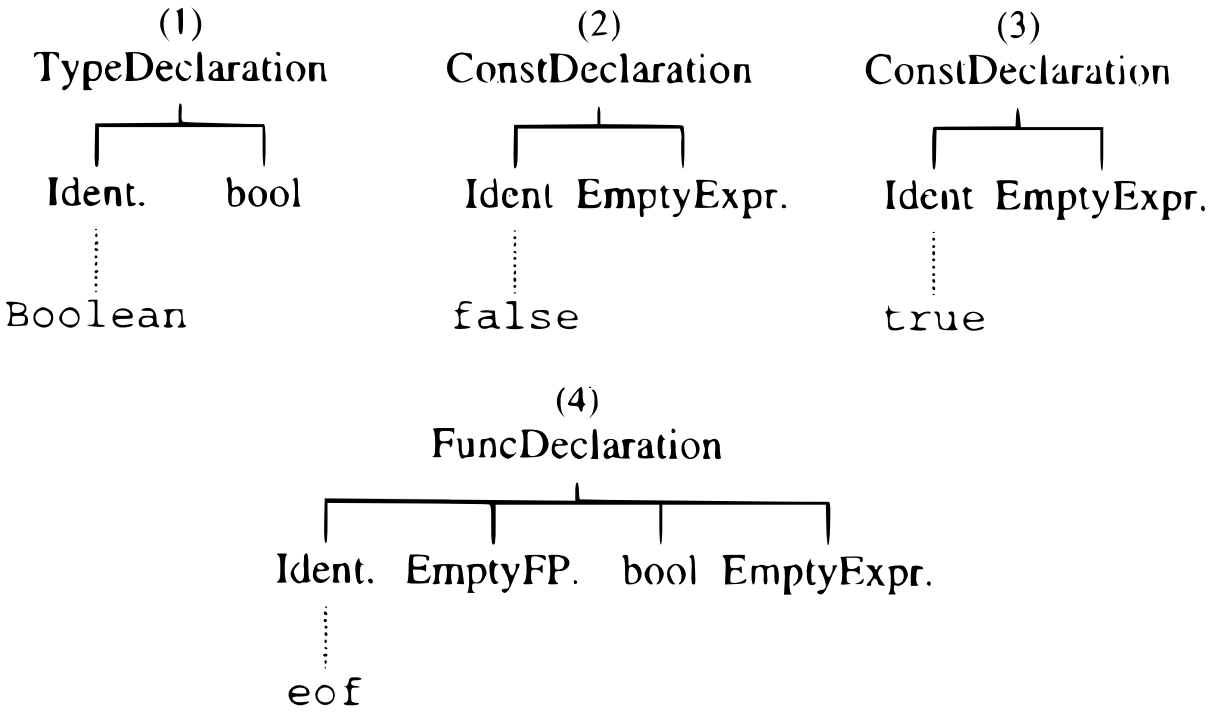
- Einlesen von Definitionen aus Quelltext
  - Ada, Haskell, VHDL, ...
- Direkt im Compiler implementiert
  - Pascal, teilweise C, Java, ...
  - (mini)-Triangle
- In beiden Fällen
  - Primitive Operationen nicht weiter in Eingabesprache beschreibbar
    ➥“black boxes”, nur Deklarationen sichtbar
- Geltungsbereich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

Standardumgebung: Realisierung in Triangle 1

- 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

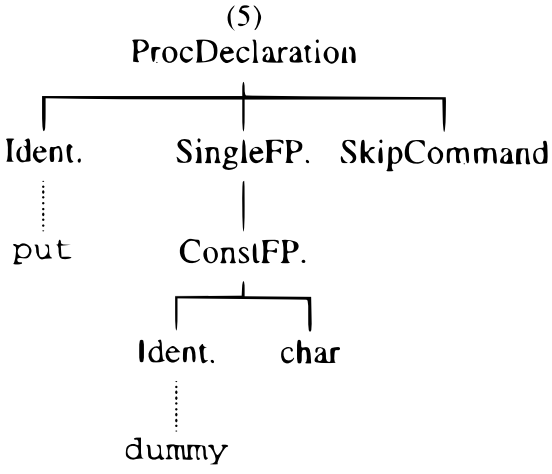
Standardumgebung: Realisierung in Triangle 2

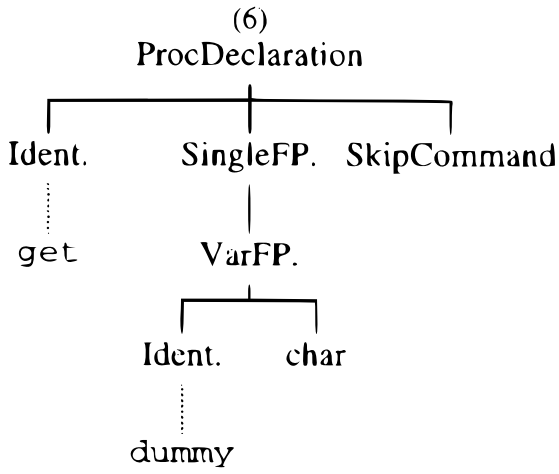
Beispiel: `Boolean`, `false`, `true`, `eof`

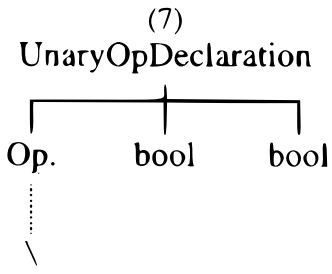


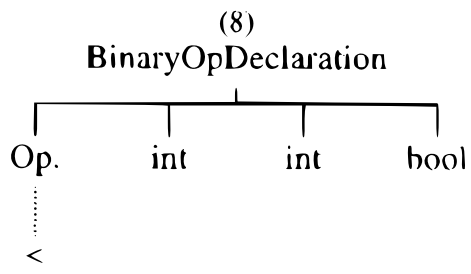
Standardumgebung: Realisierung in Triangle 3

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









Standardumgebung: Realisierung in Triangle 3

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);
```

Standardumgebung: Realisierung in Triangle 4

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;
}
```

Typäquivalenz 1

Mini-Triangle: Nur primitive Typen

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

Typäquivalenz 2

Triangle: 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?

Typäquivalenz 3

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?

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

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.

Beispiele Typäquivalenz 1

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

Beispiele Typäquivalenz 2

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

Beispiele Typäquivalenz 3

Beispiel 3

```
type Word ~ array 8 of Char;

var w1 : Word;
var w2 : Word;

if w1 = w2 then ...
```

Struktur äquivalent, Namen äquivalent

Handhabung komplexer Typen 1

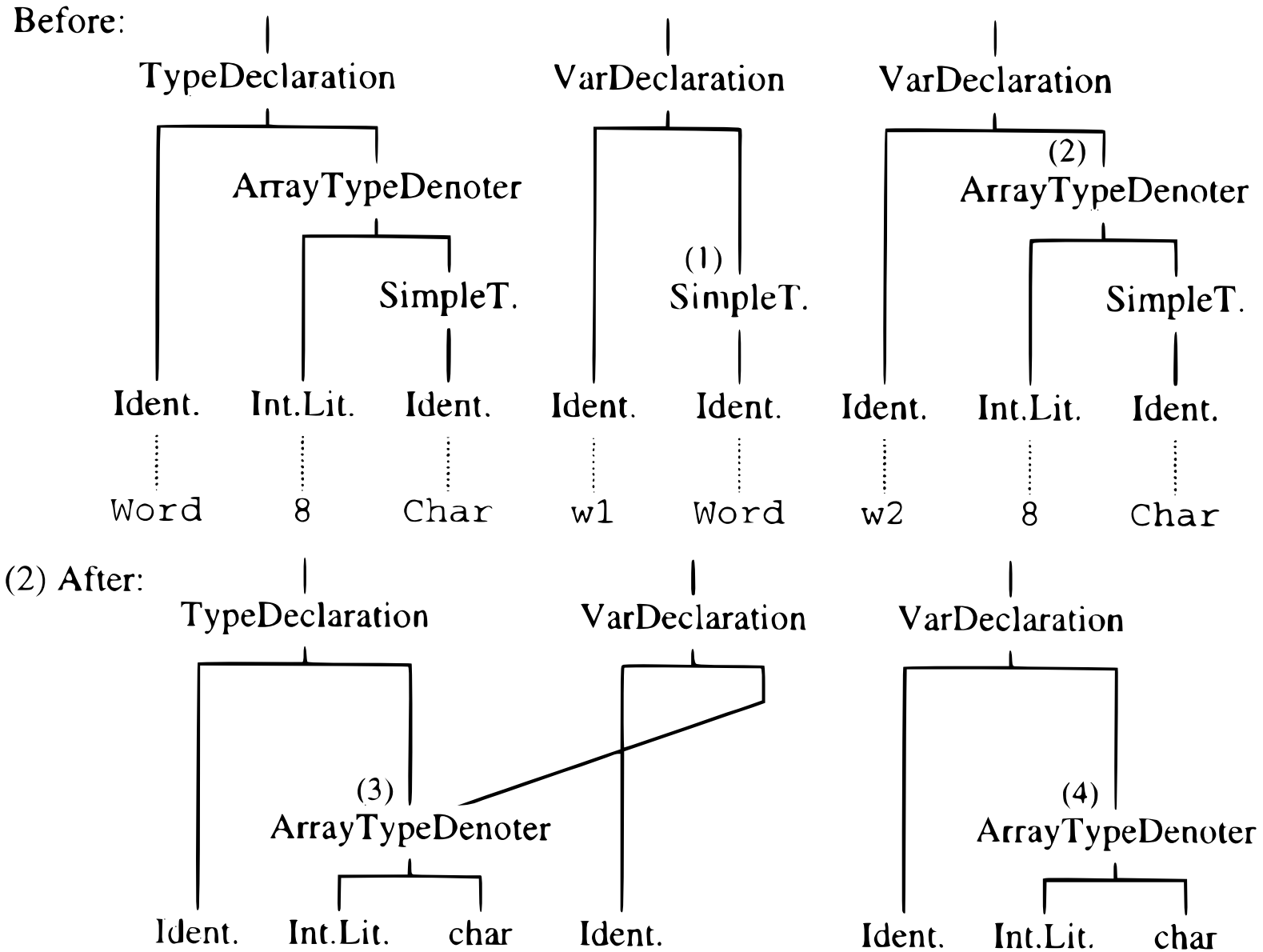
- Einfache Klasse `Type` reicht nicht mehr aus
- Kann beliebig kompliziert werden
- Idee: Verweis auf Typbeschreibung im AST
- Abstrakte Klasse `TypeDenoter`, Unterklassen
  - `IntegerTypeDenoter`
  - `ArrayTypeDenoter`
  - `RecordTypeDenoter`
  - ...

Handhabung komplexer Typen 2

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



Zusammenfassung

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

- Nächste Aufgabe: Einfache AST-Operationen