# 4. Semantikanschluss

## 4.1 Attributierte Grammatiken (prozedural)

## 4.2 Attributierte Grammatiken nach Knuth


Hanspeter Mössenböck, Institut für Systemsoftware, JKU

# *Semantikanschluss*


Parser prüft nur syntaktische Korrektheit, führt aber keine Übersetzung durch

**Aufgaben des Semantikanschlusses**

- **Prüfung von Kontextbedingungen**
  - Ist jeder verwendete Name deklariert?
  - Haben Operanden die richtigen Typen?
- **Symbollistenverwaltung**
  - Verwalten deklarierter Namen
  - Speichern von Typstrukturen
  - Verwalten von Gültigkeitsbereichen
- **Aufruf von Codeerzeugungsmethoden**

Semantische Aktionen werden beim rekursiven Abstieg in den Parser integriert und mit *attributierten Grammatiken* beschrieben


# 4. Semantikanschluss

# *Semantische Aktionen*


**Bisher: Analyse der Eingabe**

```
Number = digit {digit}.
```

Parser prüft, ob eine Eingabe syntaktisch korrekt ist
(*Number* wird hier nicht als Teil der lex. Analyse betrachtet)

**Jetzt: *Übersetzung* der Eingabe (Semantische Verarbeitung)**

Zum Beispiel: Zählen der Ziffern einer Zahl

```
Number =
   digit      (. int n = 1; .)
   { digit    (. n++; .)
   }          (. System.out.println(n); .)
   .
```

"Übersetzung" hier:

"123" ⇒ 3
"4711" ⇒ 4
"9" ⇒ 1

*Semantische Aktionen*

- beliebige Java-Anweisungen zwischen **(.** und **.)**
- werden vom Parser an der Stelle ausgeführt, an der sie in der Grammatik vorkommen
- tabellarische Schreibweise:
  Syntax links -- semantische Aktionen rechts

# *Attribute*


**Syntaxsymbole können Werte liefern (Ausgangsparameter)**

```
digit <↑val>
```

*digit* liefert als Ausgangsattribut seinen Ziffernwert (0..9)

**Attribute sind bei der Übersetzung nützlich**

Zum Beispiel: Berechnung des Werts einer Zahl

```
Number                  (. int val, n; .)
=  digit <↑val>
   { digit <↑n>         (. val = 10 * val + n; .)
   }                    (. System.out.println(val); .)
   .
```

"Übersetzung" hier:

| | | |
|---|---|---|
| "123" | ⇒ | 123 |
| "4711" | ⇒ | 4711 |
| "9" | ⇒ | 9 |

# *Eingangsattribute*


**Nonterminalsymbole können auch Eingangsattribute haben**
(Parameter, die vom Rufer mitgegeben werden)

```
Number <↓base, ↑val>
```

*base*: Ziffernbasis (z.B. 10 oder 16)
*val*: gelieferter Wert der Zahl

**Beispiel**

```
Number <↓base, ↑val>     (. int base, val, n; .)
=  digit <↑val>
   { digit <↑n>          (. val = base * val + n; .)
   }.
```

"Übersetzung" hier:

$"123"_{dec} \Rightarrow 123$

$"123"_{hex} \Rightarrow 291 = 1*16^2 + 2*16^1 + 3*16^0$

# *Attributierte Grammatiken*


Notation zur Beschreibung von Übersetzungsprozessen

Bestehen aus 3 Teilen

## 1. Syntaxregeln in EBNF

```
IdentList = ident {"," ident}.
```

## 2. **Attribute** (Parameter von Syntaxsymbolen)

```
ident<↑name>
IdentList<↓type>
```

Ausgangsattribute (*synthesized*): liefern Übersetzungsergebnis
Eingangsattribute (*inherited*): stellen Kontext bereit

## 3. Semantische Aktionen

```
(.   ... beliebige Java-Anweisungen ...   .)
```

# *Beispiel*

## ATG zur Verarbeitung von Deklarationen

```
VarDecl                          (. Struct type; .)
=  Type <↑type>
   IdentList <↓type>
   ";" .
```

```
IdentList <↓type>                (. Struct type; String name; .)
=  ident <↑name>                 (. Tab.insert(name, type); .)
   { "," ident  <↑name>          (. Tab.insert(name, type); .)
   } .
```

## Wird folgendermaßen in Parsercode umgesetzt

```
private static void VarDecl() {
   Struct type;
   type = Type();
   IdentList(type);
   check(semicolon);
}
```

ATGs sind kürzer und
lesbarer als Parsercode

```
private static void IdentList(Struct type) {
   String name;
   check(ident); name = t.val;
   Tab.insert(name, type);
   while (sym == comma) {
      scan();
      check(ident); name = t.val;
      Tab.insert(name, type);
   }
}
```

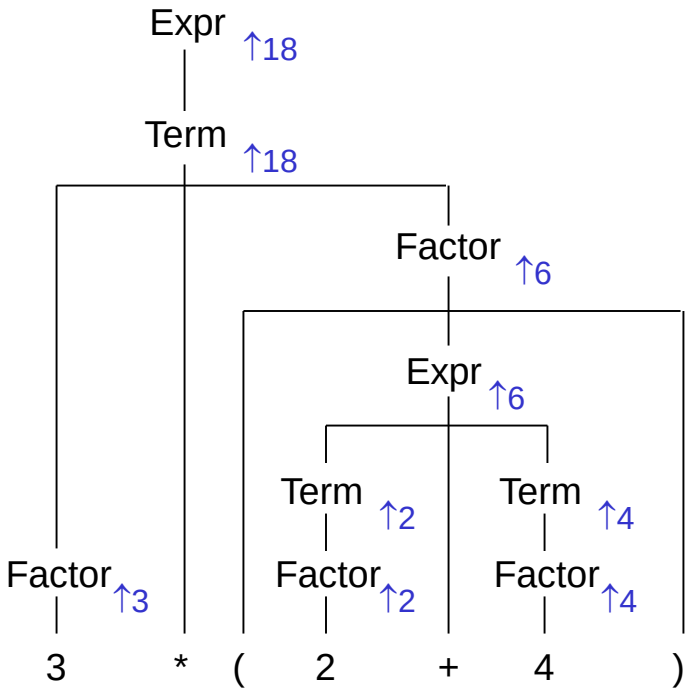
# *Beispiel: Berechnung von Konstantenausdrücken*

Eingabesatz: 3 * (2 + 4)
Übersetzungsergebnis: 18

```
Expr<↑val>                  (. int val, val1; .)
=  Term <↑val>
   { "+" Term <↑val1>       (. val = val + val1; .)
   |  "-" Term <↑val1>      (. val = val - val1; .)
   }.

Term <↑val>                 (. int val, val1; .)
=  Factor <↑val>
   { "*" Factor <↑val1>     (. val = val * val1; .)
   |  "/" Factor <↑val1>    (. val = val / val1; .)
   }.

Factor <↑val>               (. int val; .)
=  number                   (. val = t.numVal; .)
|  "(" Expr<↑val>  ")".
```



# *Umsetzung in Parsercode*


## Produktion

```
Expr <↑val>                 (. int val, val1; .)
=  Term <↑val>
   { "+" Term <↑val1>       (. val = val + val1; .)
   |  "-" Term <↑val1>      (. val = val - val1; .)
   }.
```

## Parser-Methode

```
private static int Expr() {
   int val, val1;
   val = Term();
   for (;;) {
      if (sym == plus) {
         scan();
         val1 = Term();
         val = val + val1;
      } else if (sym == minus) {
         scan();
         val1 = Term();
         val = val - val1;
      } else break;
   }
   return val;
}
```

| | | |
|---|---|---|
| Eingangsattribute | ⇒ | Parameter |
| Ausgangsattribute | ⇒ | Funktionswert (bei mehreren Ausgangsattributen Objekt verwenden) |
| Sem. Aktionen | ⇒ | eingebetteter Java-Code |

Terminalsymbole haben keine Eingangsattribute.

Bei uns haben sie auch keine Ausgangsattribute, sondern ihr Wert wird aus *t.val* oder *t.numVal* entnommen.

# *Beispiel: Verkaufsstatistik*


**ATGs sind auch außerhalb des Compilerbaus nützlich**

Beispiel: Datei mit Verkaufszahlen

```
3451    2 5 3 7 ;
3452    4 8 1 ;
3453    1 1 ;
...
```

Gewünschte Ausgabe:

```
3451    17
3452    13
3453    2
...
```

Beschreibung durch eine Grammatik

```
File    = {Article}.
Article = Code {Amount} ";".
Code    = number.
Amount  = number.
```

Immer wenn die Eingabe syntaktisch strukturiert ist, kann man sie mit ATGs verarbeiten

# *ATG für Verkaufsstatistik*

```
File                                (. int code, amount; .)
= { Article <↑code, ↑amount>        (. print(code + " " + amount); .)
  }.

Article <↑code, ↑amount>          (. int code, x, amount = 0; .)
= Number <↑code>
  { Number <↑x>                     (. amount += x; .)
  }
  ";".

Number <↑x>                         (. int x; .)
  = number                          (. x = t.numVal; .)
  .
```

*Parsercode*

```
private static void File() {
   while (sym == number) {
      ArtInfo a = Article();
      print(a.code + " " + a.amount);
   }
}
```

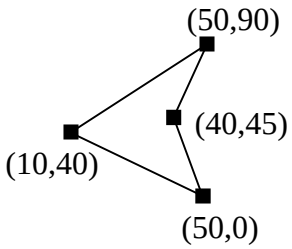
```
class ArtInfo {
   int code, amount;
}
```

```
private static ArtInfo Article() {
   ArtInfo a = new ArtInfo();
   a.code = Number();
   while (sym == number) {
      int x = Number();
      a.amount += x;
   }
   check(semicolon);
   return a;
}
```

```
private static int Number() {
   check(number);
   return t.numVal;
}
```

Terminalsymbole
- number
- semicolon
- eof

# *Beispiel: Bildbeschreibungssprache*




*beschrieben durch:*

```
POLY
   (10,40)
   (50,90)
   (40,45)
   (50,0)
END
```

*Grammatik der Eingabe:*

```
Polygon = "POLY" Point {Point} "END".
Point = "(" number "," number ")".
```

**Gesucht: Lesen der Eingabe und Zeichnen des Polygons**

```
Polygon                 (. Pt p, q; .)
=  "POLY"
   Point<↑p>            (. Turtle.start(p); .)
   { Point<↑q>          (. Turtle.move(q); .)
   }
   "END"                (. Turtle.move(p); .)
   .

Point<↑p>               (. Pt p = new Pt(); .)
=  "(" number           (. p.x = t.numVal; .)
   "," number           (. p.y = t.numVal; .)
   ")"
   .
```

*Zeichnen durch Turtle-Grafik*

| | |
|---|---|
| Turtle.start(p); | setzt den Stift auf Punkt *p* |
| Turtle.move(q); | bewegt den Stift nach *q* und zeichnet dabei eine Linie |

*Darstellung von Punkten*

```
class Pt {
   int x;
   int y;
}
```

# *Beispiel: Transformation von Infix nach Postfix*

Arithmetischer Ausdruck soll von Infix- nach Postfix-Notation übersetzt werden

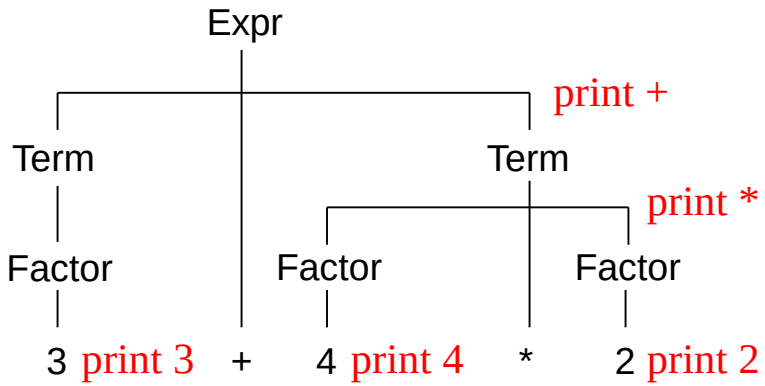
3 + 4 * 2 ⇒ 3 4 2 * +

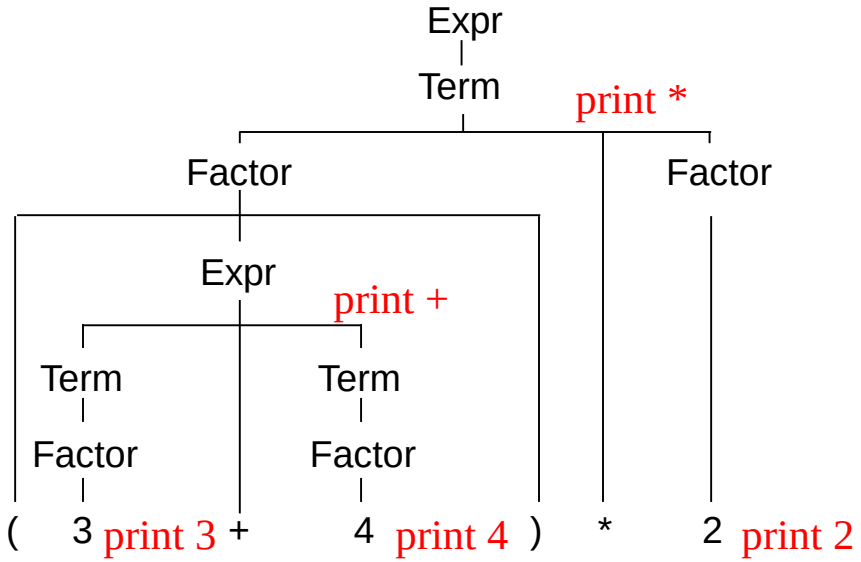
(3 + 4) * 2 ⇒ 3 4 + 2 *

```
Expr
= Term
   { "+" Term    (. print("+"); .)
   |  "-" Term   (. print("-"); .)
   }.

Term
= Factor
   { "*" Factor  (. print("*"); .)
   |  "/" Factor (. print("/"); .)
   }.

Factor
= number       (. print(t.numVal); .)
|  "(" Expr ")".
```




# 4. Semantikanschluss

## 4.1 Attributierte Grammatiken (prozedural)

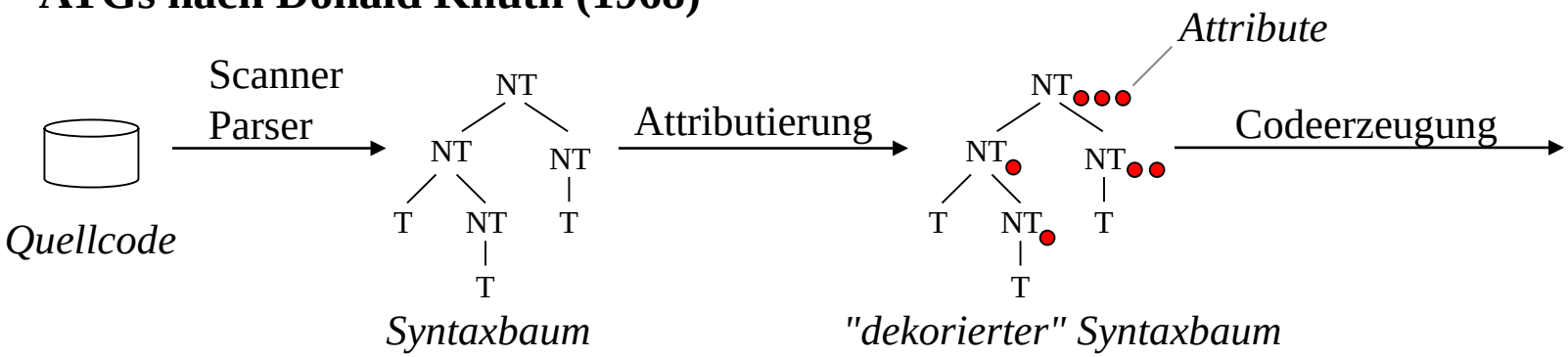
## 4.2 Attributierte Grammatiken nach Knuth

# *Idee*

**ATGs bisher**: Prozedurale Beschreibung (Übersetzungsalgorithmus)

- Jede Produktion wird von links nach rechts abgearbeitet
- Dabei werden Attribute berechnet und semantische Aktionen ausgeführt

**ATGs nach Donald Knuth (1968)**




**Nonterminalsymbole haben Attribute**

- statische Eigenschaften (ändern sich nach der Berechnung nicht mehr)
- Beispiel: Typ eines Ausdrucks, Adresse einer Variablen, ...

**Attributierung**

Durchlaufen des Syntaxbaums (u.U. mehrmals auf und ab), bis alle Attribute berechnet sind.

# *Attributierungsregeln*

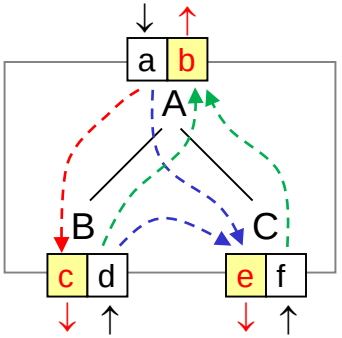
**Definieren für jede Produktion**

- *Eingangsattribute* der Symbole auf der *rechten Seite* der Produktion
- *Ausgangsattribute* der Symbole auf der *linken Seite* der Produktion
- *Kontextbedingungen* (wenn nötig)

**Beispiel**

Produktion *p*:

$A_{\downarrow a \uparrow b} = B_{\downarrow c \uparrow d}\ C_{\downarrow e \uparrow f}\ .$




*Produktion p* stellt einen Ausschnitt des Syntaxbaums dar

Es müssen alle Attribute definiert werden, die *p* verlassen (d.h. *b*, *c*, *e*)

Attributierungsregeln *R*(*p*) zum Beispiel:

```
c = a;
e = d + foo(a);
b = d + f;
```

Kontextbedingung *CC*(*p*) zum Beispiel:

```
d >= f
```

# *Beispiel: Berechnung des Werts einer Hex-Zahl*

**Grammatik** (muss in BNF vorliegen, damit man einen Syntaxbaum aufbauen kann)

```
Number = Digits.          // Dezimalzahl
Number = Digits "H".      // Hexadezimalzahl
Digits = hex.             // hex ... 0..9, A..F
Digits = Digits hex.
```

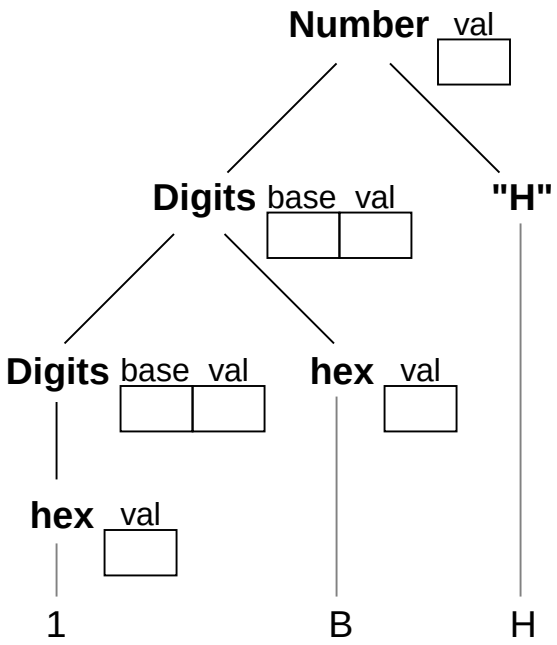
**Attribute**

Number ↑val

Digits ↓base ↑val

hex ↑val

**Syntaxbaum** für die Eingabe: 1BH




Attribute sind noch nicht berechnet

# *Attributierungsregeln*


Number = Digits.
Number = Digits "H".
Digits = hex.
Digits = Digits hex.

**Produktion 1**

| $Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$. |
|---|
| Digits.base = 10;<br>Number.val = Digits.val; |

**Produktion 2**

| $Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H". |
|---|
| Digits.base = 16;<br>Number.val = Digits.val; |

**Produktion 3**

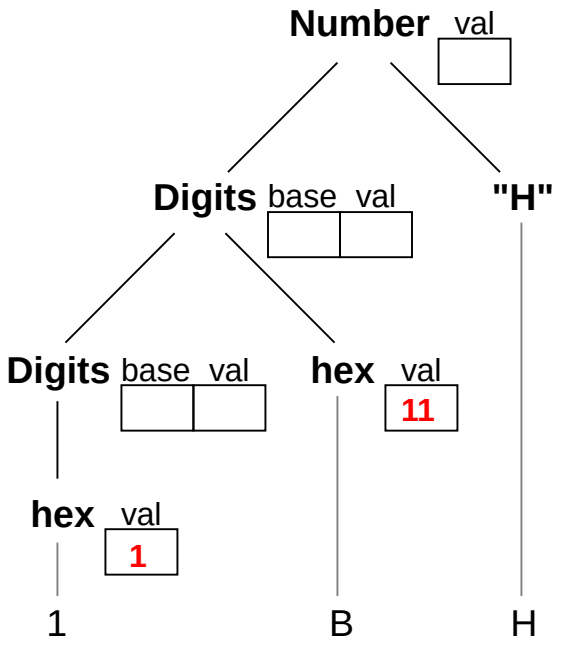
| $Digits_{\downarrow base \uparrow val} = hex_{\uparrow val}$. |
|---|
| Digits.val = hex.val;<br>CC: Digits.base == 10 && 0 ≤ hex.val ≤ 9<br>\|\| Digits.base == 16 && 0 ≤ hex.val ≤ 15 |

**Produktion 4**

| $Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$. |
|---|
| Digits1.base = Digits.base;<br>Digits.val = Digits1.val * Digits.base + hex.val;<br>CC: Digits.base == 10 && 0 ≤ hex.val ≤ 9<br>\|\| Digits.base == 16 && 0 ≤ hex.val ≤ 15 |

# *Attributierung des Baums*

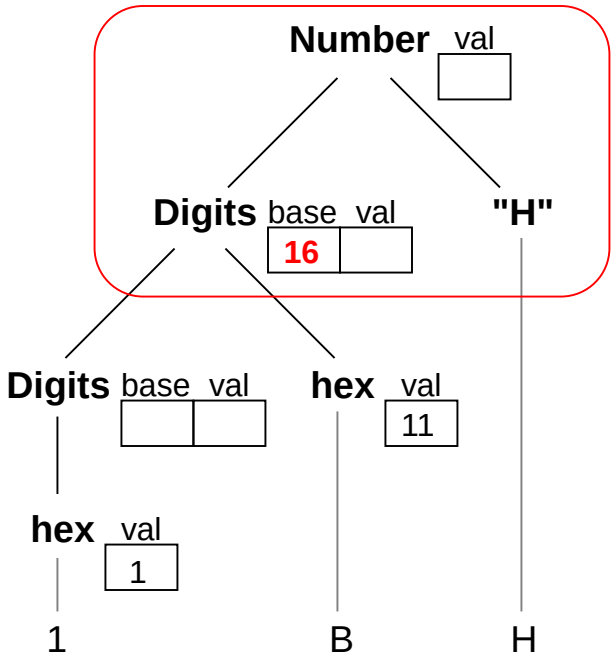
**Scanner füllt die Attributwerte der Terminalsymbole**



# *Attributierung des Baums (Forts.)*

**Durchlauf des Baums von oben nach unten**

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind




**Produktion 2**

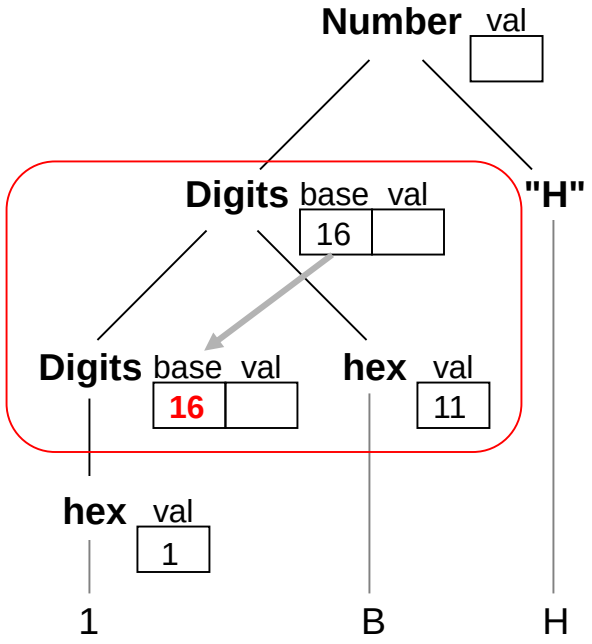
$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H".

Digits.base = 16;
Number.val = Digits.val;

# *Attributierung des Baums (Forts.)*

**Durchlauf des Baums von oben nach unten**

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind




**Produktion 4**

$Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$.

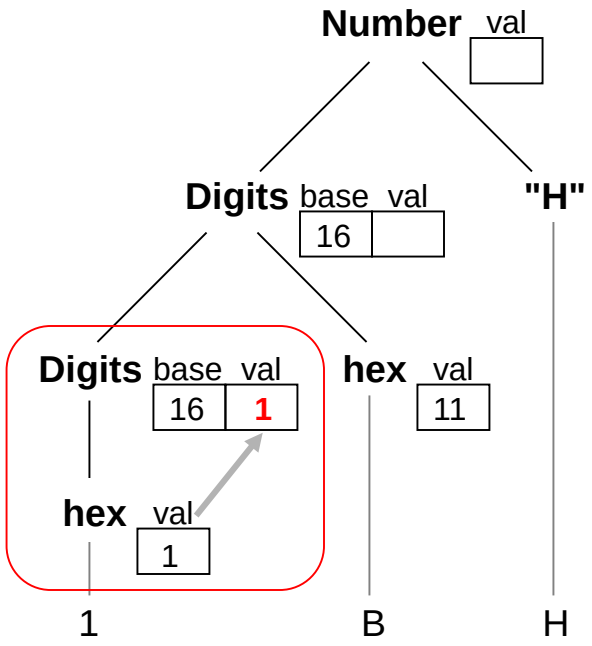
Digits1.base = Digits.base;
Digits.val = Digits1.val * Digits.base + hex.val;

CC: Digits.base == 10 && $0 \le$ hex.val $\le 9$
|| Digits.base == 16 && $0 \le$ hex.val $\le 15$

# *Attributierung des Baums (Forts.)*

**Durchlauf des Baums von oben nach unten**

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind




**Produktion 3**

$Digits_{\downarrow base \uparrow val} = hex_{\uparrow val}$.

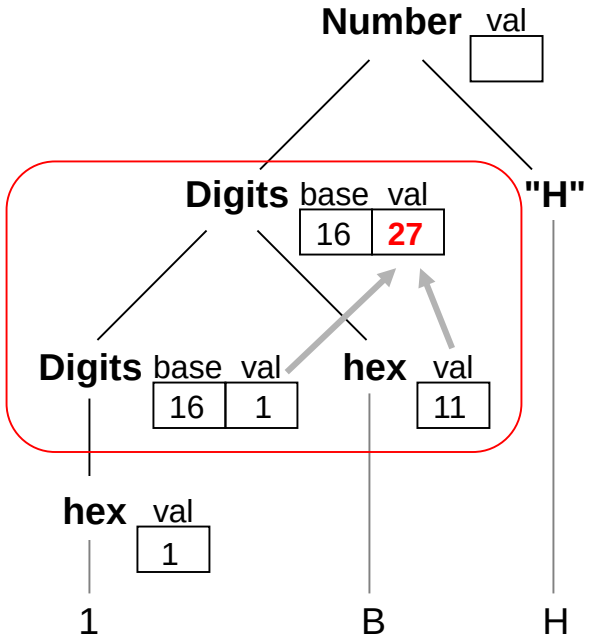
Digits.val = hex.val;

CC: Digits.base == 10 && $0 \leq$ hex.val $\leq 9$
|| Digits.base == 16 && $0 \leq$ hex.val $\leq 15$

# *Attributierung des Baums (Forts.)*

**Durchlauf des Baums von unten nach oben**

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind




**Produktion 4**

$Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$.

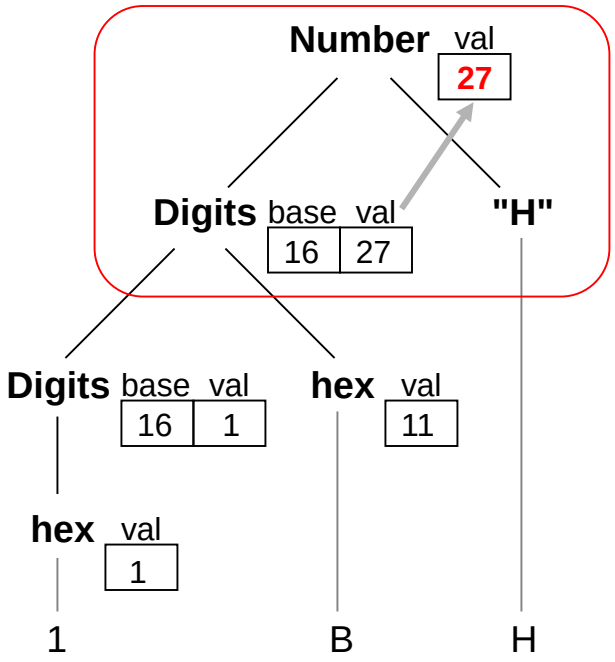
Digits1.base = Digits.base;
Digits.val = Digits1.val * Digits.base + hex.val;

CC: Digits.base == 10 && $0 \le$ hex.val $\le 9$
|| Digits.base == 16 && $0 \le$ hex.val $\le 15$

# *Attributierung des Baums (Forts.)*

**Durchlauf des Baums von unten nach oben**

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind




**Produktion 2**

$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H".

Digits.base = 16;
Number.val = Digits.val;

**Alle Attribute sind berechnet ⇒ Ende des Durchlaufs**

# *Definition einer ATG nach Knuth*


**Attributierte Grammatik**

ATG = (CFG, A, R, CC)

| | |
|---|---|
| CFG ... | kontextfreie Grammatik |
| A ... | Menge von Attributen |
| R ... | Menge von Attributierungsregeln |
| CC ... | Menge von Kontextbedingungen |

**Kontextfreie Grammatik**

CFG = (T, N, P, S)

| | |
|---|---|
| T ... | Terminalsymbole |
| N ... | Nonterminalsymbole |
| P ... | Produktionen |
| S ... | Startsymbol |

**Attribute**

A(X) ... Attribute des Symbols X (geschrieben als X.a, X.b, ...)
- AS(X) ... Ausgangsattribute von X (synthesized)
- AI(X) ... Eingangsattribute von X (inherited)

**Attributierungsregeln**

R(p) ... Attributierungsregeln für Produktion p: X0 = X1 ... Xn
- R(p) = {Xi.a = f(Xj.b, ..., Xk.c)}
- für alle Ausgangsattribute der linken und alle Eingangsattribute der rechten Seite von p

**Kontextbedingungen**

CC(p) ... Kontextbedingungen für Produktion p: X0 = X1 ... Xn
- in Form eines Booleschen Ausdrucks B(Xi.a, ..., Xj.b)
- prüfen die "statische Semantik", d.h. ob die Eingabe semantisch korrekt ist

# *Vollständige ATGs*


**Definition**

Eine ATG heißt *vollständig*, wenn für alle Produktionen
p: X = Y1 ... Yn
gilt: alle AS(X) und alle AI(Yi) werden in R(p) eingestellt

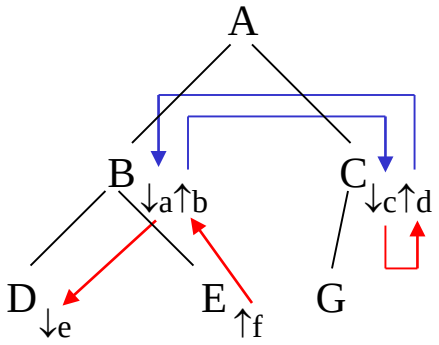
# *Wohlgeformte ATGs (WAGs)*

**Definition**

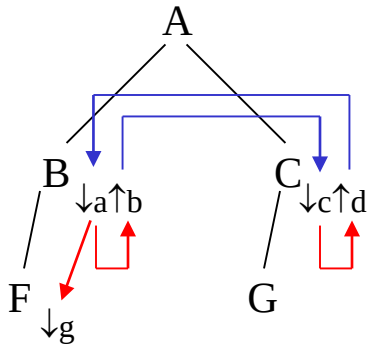
Eine ATG heißt *wohlgeformt* (well-defined, WAG)
- wenn die ATG vollständig ist und
- wenn die Attributbeziehungen in *jedem möglichen Syntaxbaum* kreisfrei sind

Mit anderen Worten:
Für jeden möglichen Syntaxbaum lässt sich eine Attributierungsreihenfolge finden

**Beispiel**




wohlgeformt




zirkulär ⇒ nicht wohlgeformt

Prüfung auf Wohlgeformtheit ist *NP-vollständig* (exponentielles Zeitverhalten)!
Es gibt aber Unterklassen von WAG, bei denen das einfacher festzustellen ist.

# *Geordnete ATGs (OAGs)*

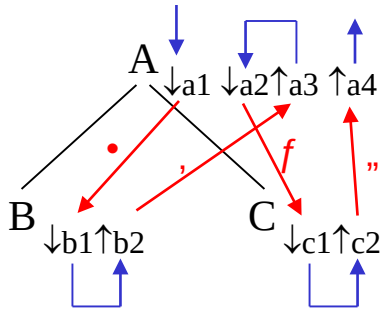

**Definition**

Eine ATG heißt *geordnet* (OAG), wenn sich *für jede Produktion* unabhängig von ihrem Kontext im Syntaxbaum eine *fixe Attributierungsreihenfolge* angeben lässt.

Attributierungscode einer Produktion *p* kann z.B. durch folgende Operationen angegeben werden:

| | | |
|---|---|---|
| $comp_i$ | ... | führe Attributierungsregel *i* in R(*p*) aus |
| up | ... | gehe zum Vater im Syntaxbaum |
| $down_i$ | ... | gehe zu Sohn *i* im Syntaxbaum |

**Beispiel**

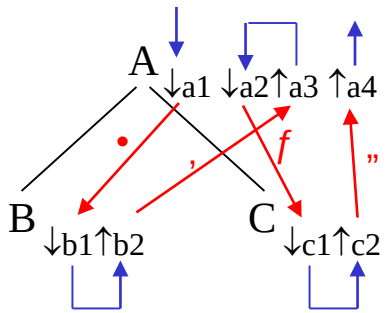



*Attributierungscode*

comp •
$down_B$
comp ,
up
comp *f*
$down_C$
comp „
up

# *Gegenbeispiel*

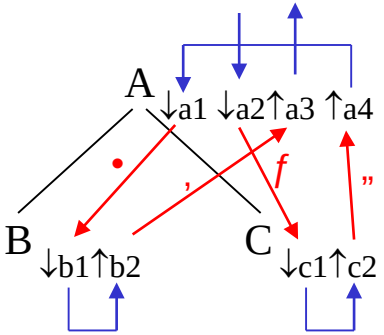
Erstes Vorkommen von *A* (wie vorhin)




*Attributierungscode*

comp •
$down_B$
comp ,
up
comp *f*
$down_C$
comp „
up

Weiteres Vorkommen von *A* im Syntaxbaum (mit anderer Attributierungsreihenfolge)




*Attributierungscode*

comp *f*
$down_C$
comp „
up
comp •
$down_B$
comp ,
up

Reihenfolge der Attributierung hängt davon ab, wo *A* im Syntaxbaum vorkommt
=> ATG ist nicht geordnet!

Prüfung, ob eine Grammatik OAG ist, kann in Polynomialzeit erfolgen.

# *L-attributierte ATGs (LAGs)*

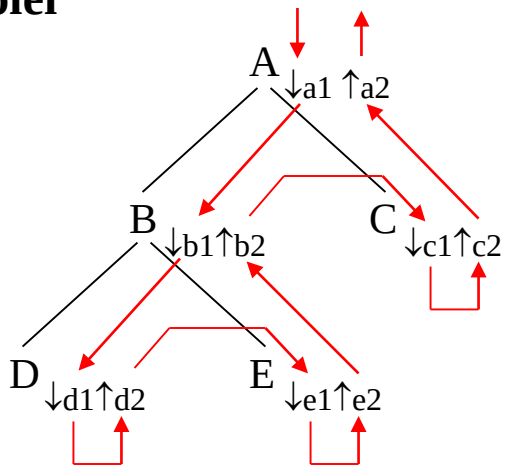

**Definition**

Eine ATG heißt *L-attributiert* (LAG), wenn sämtliche Attribute im Syntaxbaum sich in einem einzigen Durchgang (runter und rauf, von links nach rechts) berechnen lassen.

*Mit anderen Worten*:

Wenn man die Attribute schritthaltend mit der Syntaxanalyse berechnen kann.
Für LAGs muss nicht einmal ein Syntaxbaum aufgebaut werden (entspricht unseren prozeduralen ATGs).

**Beispiel**




Information, die weiter vorne im Programm steht, kann nach hinten transportiert werden, aber nicht umgekehrt!

# *Beziehungen zwischen ATG-Klassen*


**ATG ⊃ WAG ⊃ OAG ⊃ LAG**

das heißt
- jede LAG ist geordnet
- jede OAG ist wohlgeformt

WAG ist mächtiger als OAG
OAG ist mächtiger als LAG