Aufnahme der Hybrid-Vorlesung

- **Hybrid-Vorlesung mit Aufnahme**
  - Die Aufnahme ist anschließend in Stud.IP verfügbar
  - Nutzen Sie die Gelegenheit zur Live-Veranstaltung!
- Wir nehmen auf
  - Folien, Dozent, Live-Audio sowie BBB-Audio
  - Ihre Stimme beim Fragen und Sprechen
  - **Durch aktive Teilnahme erklären Sie sich einverstanden!**
- Fragen: Live, im Chat, Sprechen in der BBB-Sitzung

Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

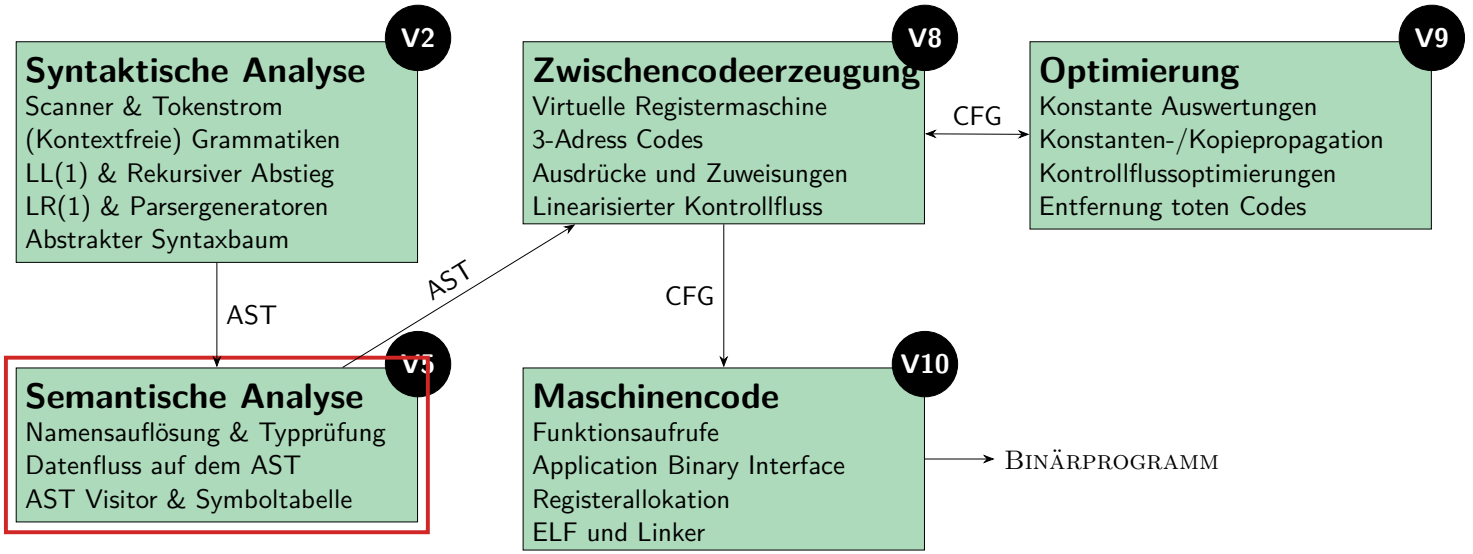
Programmiersprachen und Übersetzer

05 - Semantische Analyse

Christian Dietrich

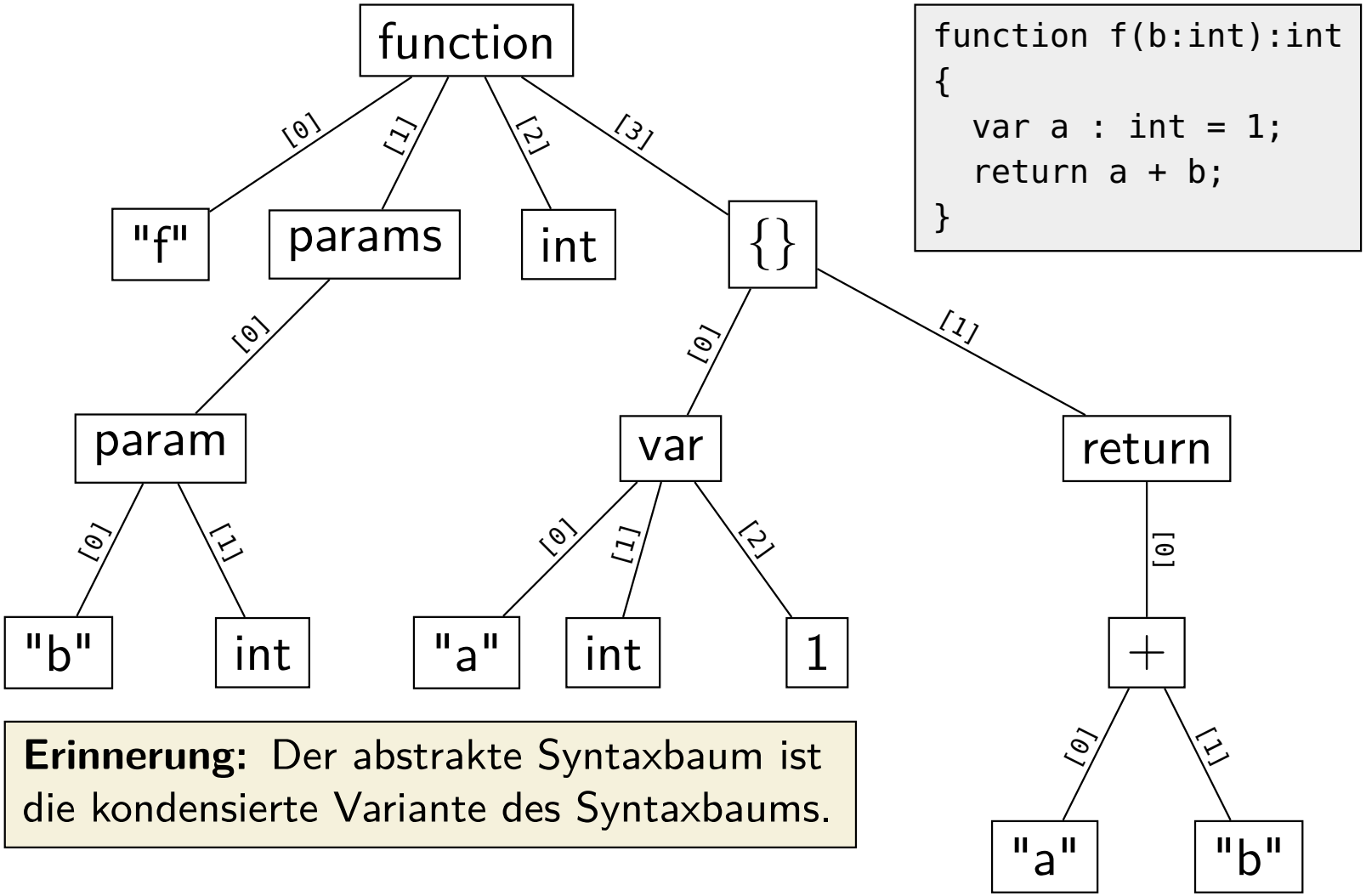
Sommersemester 2024

Einordnung in die Vorlesung: Semantische Analyse



- Übersetzer prüft bei der semantische Analyse die letzten Sprachregeln.
  $\Rightarrow$ In der semantischen Analyse werden die letzten Fehler entdeckt.
- Was sollte der *effektive* und *effiziente* Informatiker darüber wissen?
  - Datenfluss auf einem Baum geschieht entlang und quer zu den Kanten.
  - Übersetzertechniken sind auch in anderen Bereichen einsetzbar.

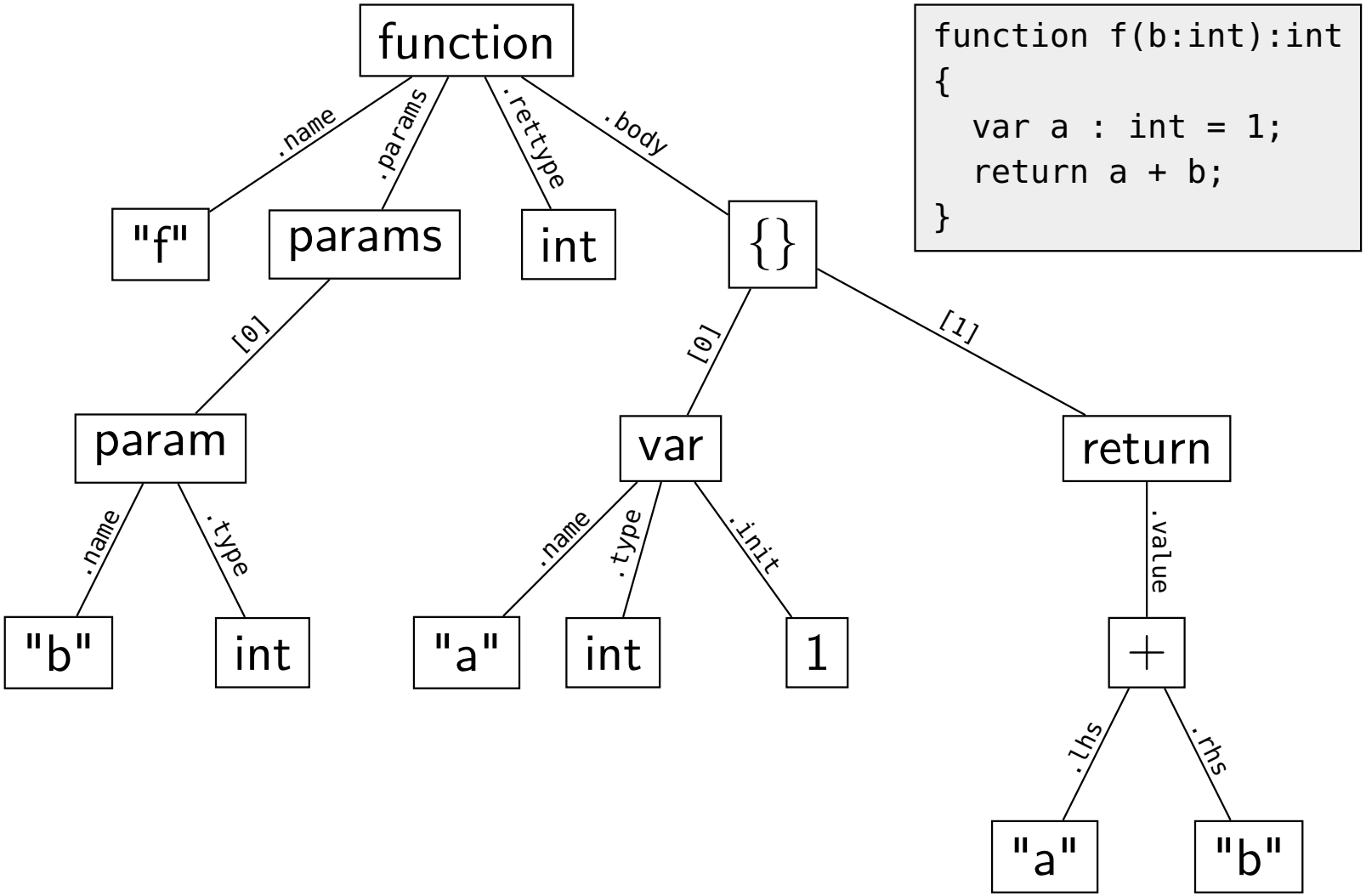
Beispiel: Semantische Analyse eines Programms




```
function f(b:int):int
{
  var a : int = 1;
  return a + b;
}
```

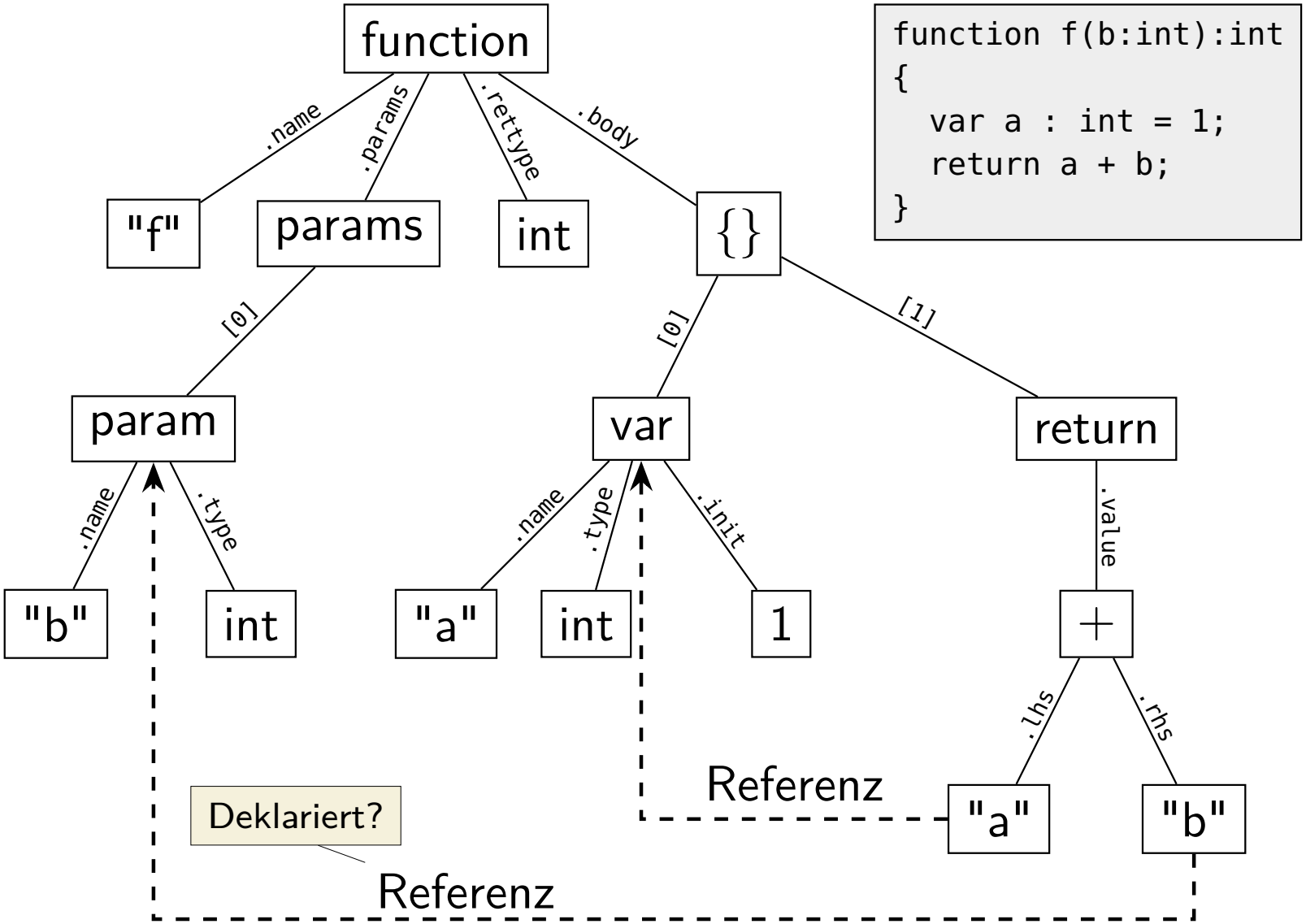
Erinnerung: Der abstrakte Syntaxbaum ist die kondensierte Variante des Syntaxbaums.

Beispiel: Semantische Analyse eines Programms

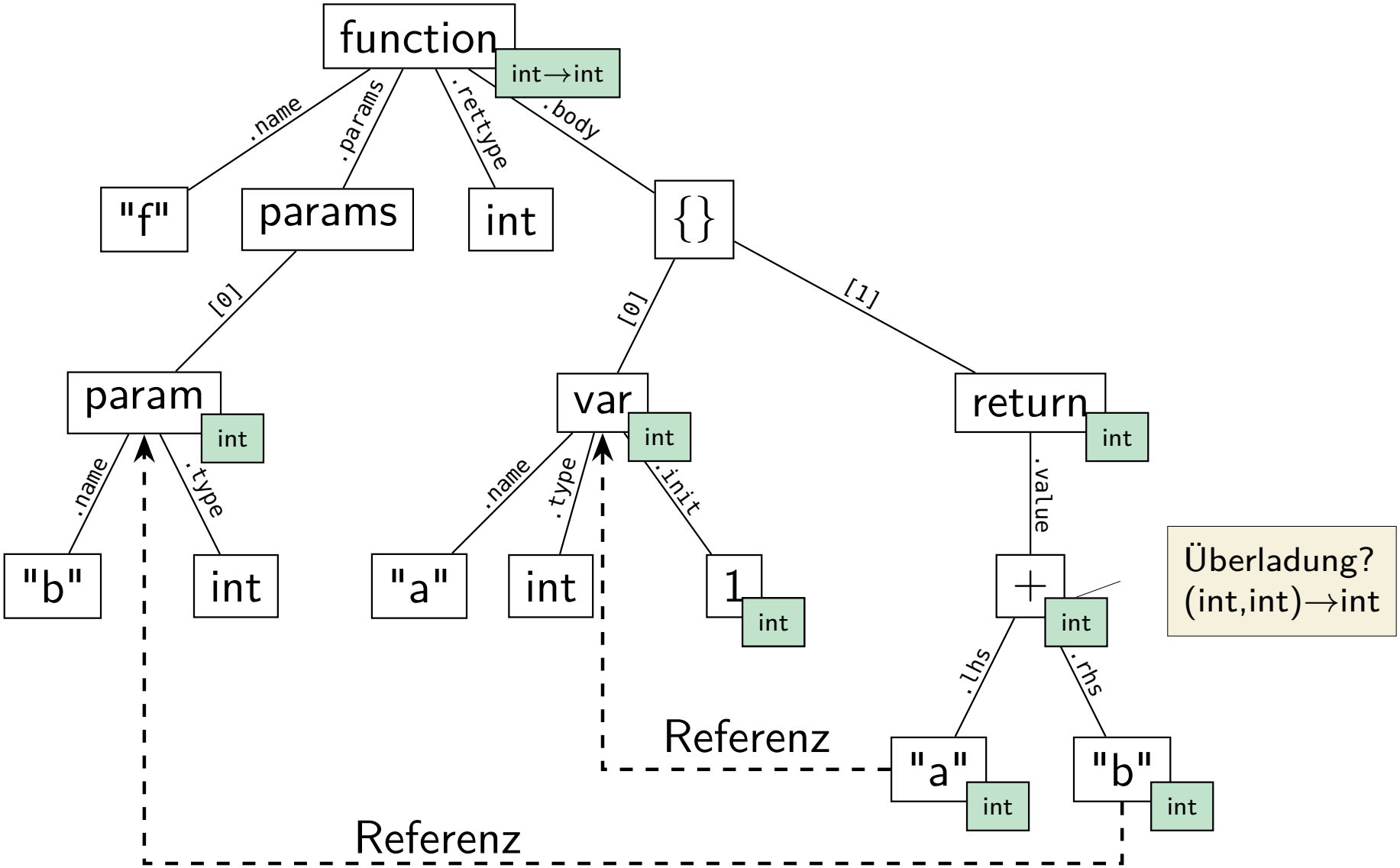



```
function f(b:int):int
{
  var a : int = 1;
  return a + b;
}
```

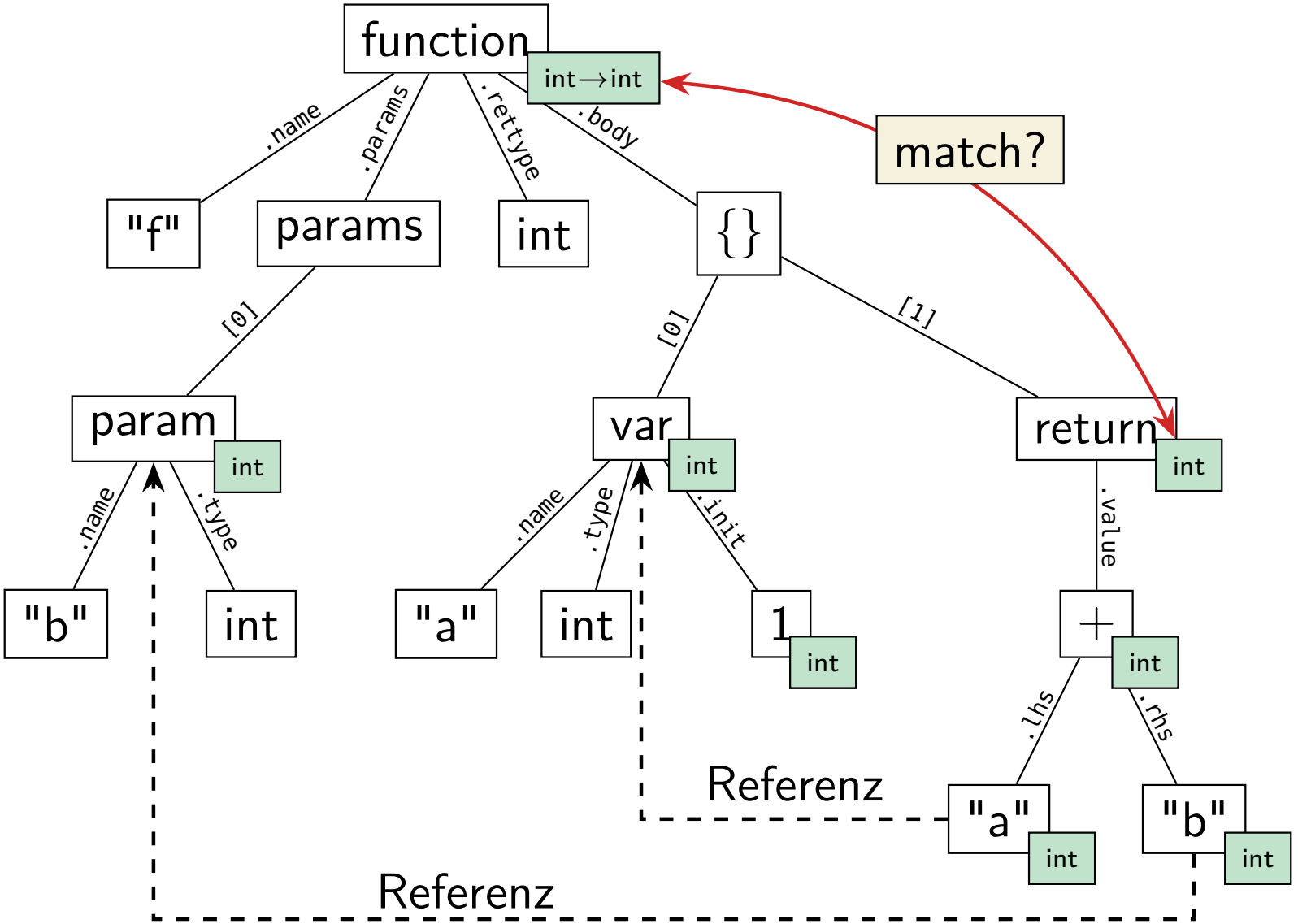
Beispiel: Semantische Analyse eines Programms



Beispiel: Semantische Analyse eines Programms



Beispiel: Semantische Analyse eines Programms



Ziele der Semantischen Analyse

Technische Universität Braunschweig

Syntaktische und semantische Analyse prüfen alle Sprachregeln

Die syntaktische Analyse prüft die kontextfreien Regeln einer Sprache. Alles, was nicht kontextfrei geprüft werden kann, überprüft die semantische Analyse.

- Abschließende Prüfung der Sprachregeln und Informationsgewinnung
  - Alle restlichen, nicht-syntaktischen, Sprachfehler werden hier aufgedeckt.
  - „Wo ist diese verwendete Variable deklariert und mit welchem Typ?“

Ziele der Semantischen Analyse

Syntaktische und semantische Analyse prüfen alle Sprachregeln

Die syntaktische Analyse prüft die kontextfreien Regeln einer Sprache. Alles, was nicht kontextfrei geprüft werden kann, überprüft die semantische Analyse.

- Abschließende Prüfung der Sprachregeln und Informationsgewinnung
  - Alle restlichen, nicht-syntaktischen, Sprachfehler werden hier aufgedeckt.
  - „Wo ist diese verwendete Variable deklariert und mit welchem Typ?“
- Deklariertheit: Kann jede (statische) Namensreferenz aufgelöst werden?
  - Zu jedem Namen(spfad) muss eine Deklaration gefunden werden.
  - Auffinden der passenden Überladung für eine Aufrufstelle
- Typkonsistenz: Werden die statischen Typen korrekt verwendet?
  - Die Typen der Argumente müssen zur Signatur einer Operation passen.
  - Nicht jede Inkonsistenz ist ein Fehler (Implizite Typumwandlung)

Ziele der Semantischen Analyse

Syntaktische und semantische Analyse prüfen alle Sprachregeln

Die syntaktische Analyse prüft die kontextfreien Regeln einer Sprache. Alles, was nicht kontextfrei geprüft werden kann, überprüft die semantische Analyse.

- Abschließende Prüfung der Sprachregeln und Informationsgewinnung
  - Alle restlichen, nicht-syntaktischen, Sprachfehler werden hier aufgedeckt.
  - „Wo ist diese verwendete Variable deklariert und mit welchem Typ?“
- Deklariertheit: Kann jede (statische) Namensreferenz aufgelöst werden?
  - Zu jedem Namen(spfad) muss eine Deklaration gefunden werden.
  - Auffinden der passenden Überladung für eine Aufrufstelle
- Typkonsistenz: Werden die statischen Typen korrekt verwendet?
  - Die Typen der Argumente müssen zur Signatur einer Operation passen.
  - Nicht jede Inkonsistenz ist ein Fehler (Implizite Typumwandlung)

AST Knoten

AST: Baumstruktur im Übersetzer

Theoretisch ganz einfach. Ein AST ist eine Menge von Knoten....

```
struct node_t {
  int    type;
  string token;
  vector<node_t> children;
};
```

```
string getName(node_t *n) {
  if (n->type == VAR_DECL) {
    return n->children[1].token;
  } else if (n->type == VAR_REF) {
    return n->token;
  } ....
}
```

AST: Baumstruktur im Übersetzer

Theoretisch ganz einfach. Ein AST ist eine Menge von Knoten....

```
struct node_t {
  int    type;
  string token;
  vector<node_t> children;
};
```

```
string getName(node_t *n) {
  if (n->type == VAR_DECL) {
    return n->children[1].token;
  } else if (n->type == VAR_REF) {
    return n->token;
  } ....
}
```

NEIN

- **In Realität**: Code des Übersetzers muss auf dem AST operieren
  - Struktur der Grammatik sollte sich in den Datenstrukturen widerspiegeln
  - Übersetzer sind komplexe Programme, Bugs sind besonders schwerwiegend
  - Nutzung der verfügbaren Sprachfeatures, um Übersetzer robust zu machen

```
class ClassDef extends Definition {
  public Name                 name;
  public ClassDef             extending;
  public List<InterfaceDef> implementing;
  public List<Definition>   defs;
  ....
}
```

Java

AST: Baumstruktur im Übersetzer

Theoretisch ganz einfach. Ein AST ist eine Menge von Knoten....

```
struct node_t {
  int    type;
  string token;
  vector<node_t> children;
};
```

```
string getName(node_t *n) {
  if (n->type == VAR_DECL) {
    return n->children[1].token;
  } else if (n->type == VAR_REF) {
    return n->token;
  } ....
}
```

NEIN

- **In Realität**: Code des Übersetzers muss auf dem AST operieren
  - Struktur der Grammatik sollte sich in den Datenstrukturen widerspiegeln
  - Übersetzer si… …gs sind besonders schwerwiegend
  - Nutzung der verfügbaren Sprachfeatures, um Übersetzer robust zu machen

Knotentypen als Klassen

```
class ClassDef extends Definition {
  public Name                  name;
  public ClassDef              extendin…
  public List<InterfaceDef> implementi…;
  public List<Definition>      defs;
  ....
}
```

Kinder haben Namen

Kinder haben Typen

Java

AST: Eine Hierarchie von Knotentypen

Technische Universität Braunschweig

- Die Grammatik legt bereits eine Hierarchie von Knotentypen nahe
  - Grammatik: *stmt* $\rightarrow$ assign_stmt | while_stmt | if_stmt | ...
  - AST-Definition: `class AssignStmt extends Stmt {...}`

AST: Eine Hierarchie von Knotentypen

Technische Universität Braunschweig

- Die Grammatik legt bereits eine Hierarchie von Knotentypen nahe
  Grammatik: *stmt* $\rightarrow$ assign_stmt | while_stmt | if_stmt | ...
  AST-Definition: `class AssignStmt extends Stmt {...}`
- Zusammenfassung von Ähnlichkeiten verschiedener Knotentypen
  Beispiel: Viele Deklarationen können einen Namen tragen. Lassen wir diese von `NamedDecl` erben, so können wir sie gleich behandeln.

AST: Eine Hierarchie von Knotentypen

- Die Grammatik legt bereits eine Hierarchie von Knotentypen nahe
  Grammatik: *stmt* → assign_stmt | while_stmt | if_stmt | …
  AST-Definition: `class AssignStmt extends Stmt {...}`
- Zusammenfassung von Ähnlichkeiten verschiedener Knotentypen
  Beispiel: Viele Deklarationen können einen Namen tragen. Lassen wir diese von `NamedDecl` erben, so können wir sie gleich behandeln.
- Häufig verwendete Klassen von Knotentypen
  - Expression: Ausdrücke, die man auswerten kann und die ein Ergebnis liefern.
    Beispiele: Addition, Dereferenzierung, `(<cond> ? <then> : <else>)`
  - Statement: Anweisungen, die sequentiell bearbeitet werden.
    Beispiele: if-else, Schleifen, Zuweisungen
  - Typen: Literal notierte Typausdrücke
    Beispiele: Typname, Typausdruck innerhalb eines Casts
  - Deklaration: Jede statische Bekanntmachung für den Compiler
    Beispiele: Variablendefinition, Klassendeklarationen, `typedef`

AST: Zusätzliche Attribute

Technische Universität Braunschweig

- Der Parser erzeugt die Knoten und die Baumstruktur.
  - Parseraktionen machen dies nebenher, wenn eine Regel angewendet wird:

```
while_stmt -> WHILE ( expr ) body  { new WhileStmt(cond=$3, body=$5) }
```

  - `expr` ist ein Expression-Knoten und ist über $3 zugreifbar.

AST: Zusätzliche Attribute

- Der Parser erzeugt die Knoten und die Baumstruktur.
  - Parseraktionen machen dies nebenher, wenn eine Regel angewendet wird:

```
while_stmt -> WHILE ( expr ) body  { new WhileStmt(cond=$3, body=$5) }
```

  - `expr` ist ein Expression-Knoten und ist über $3 zugreifbar.
- Semantische Analyse berechnet zusätzliche Attribute
  - Datenhaltung während Analyse, Wissensbasis für den Rest des Übersetzers
  - *Beispiel*: Berechneten Typ einer Expression, Zeiger auf die Deklaration

AST: Zusätzliche Attribute

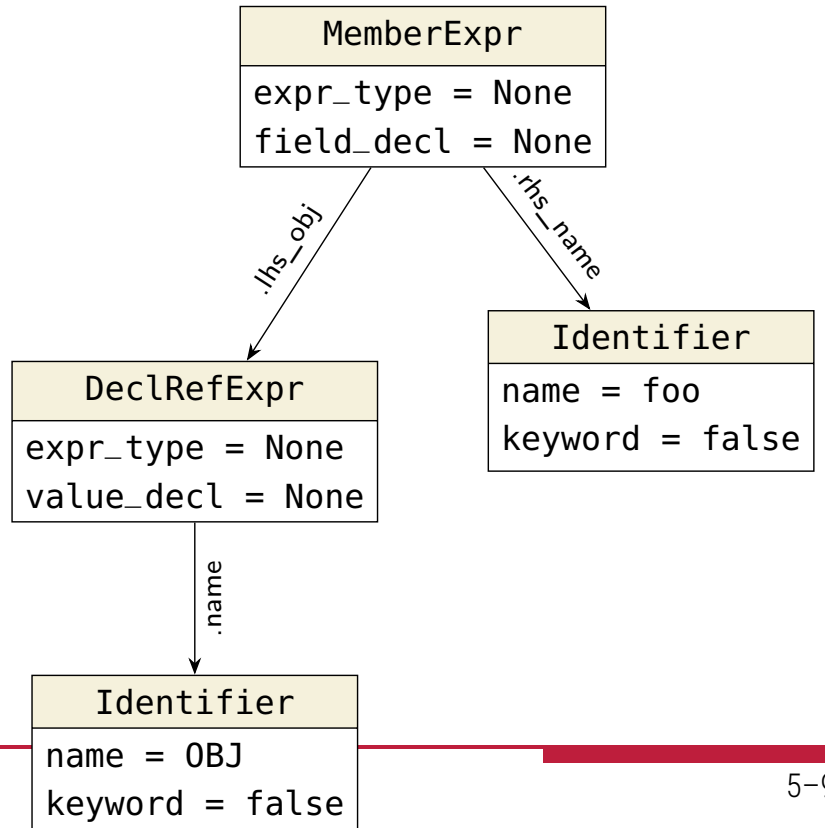
- Der Parser erzeugt die Knoten und die Baumstruktur.
  - Parseraktionen machen dies nebenher, wenn eine Regel angewendet wird:

```
while_stmt -> WHILE ( expr ) body  { new WhileStmt(cond=$3, body=$5) }
```

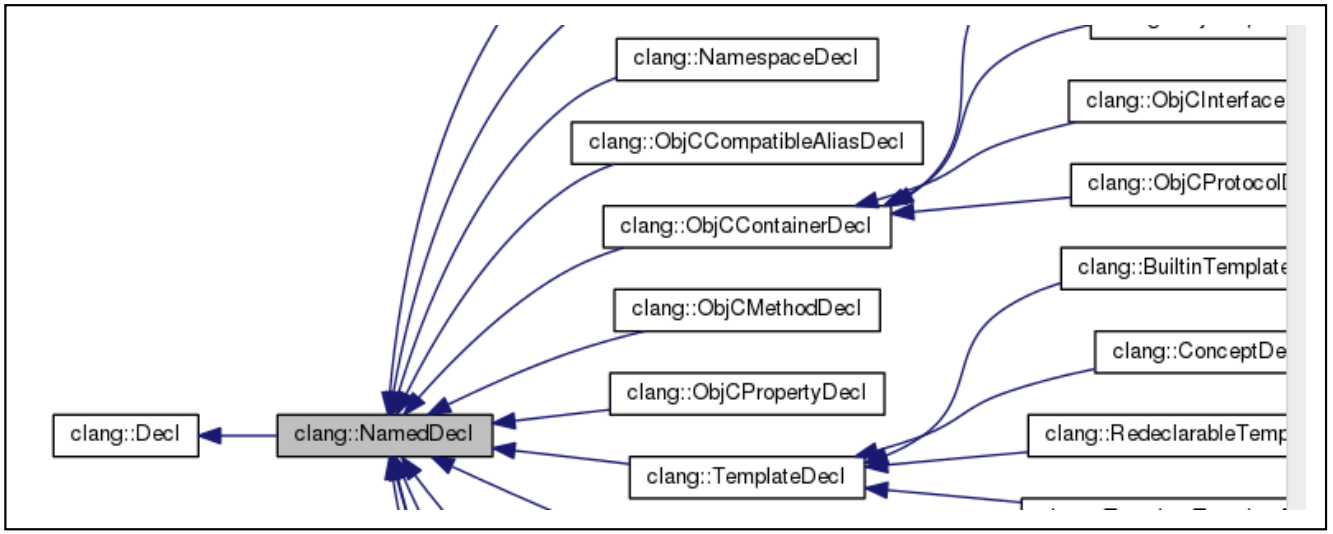
  - `expr` ist ein Expression-Knoten und ist über $3 zugreifbar.
- Semantische Analyse berechnet zusätzliche Attribute
  - Datenhaltung während Analyse, Wissensbasis für den Rest des Übersetzers
  - *Beispiel*: Berechneten Typ einer Expression, Zeiger auf die Deklaration

```python
class MemberExpr(Expr):
  # AST Kinder:
  lhs_obj    : Expr
  rhs_name   : Identifier

  # AST Attribute
  expr_type  : Type       = None
  field_decl : FieldDecl  = None

  def __init__(self, L, R):
    self.lhs_obj   = L
    self.rhs_name  = R
  ...
```

Python



Clang: AST-Knoten und ihre Attribute



Ausschnitt aus der Klassenhierarchie

Public Member Functions

IdentifierInfo *	**getIdentifier** () const Get the identifier that names this declaration, if there is one. More...
StringRef	**getName** () const Get the name of identifier for this declaration as a StringRef. More...
std::string	**getNameAsString** () const Get a human-readable name for the declaration, even if it is one of the special kinds of names (C++ constructor, Objective-C selector, etc). More...

Öffentliches Interface von `clang::NamedDecl`

Clang: AST für ein einfaches Programm

```c
int f(int x) {
    int result = (x / 42);
    return result;
}
```
C

- Clang: C/C++-Frontend für LLVM
- Source-to-Source-Transformation
- `clang::VarDecl` ist ein `clang::NamedDecl`

$ clang -Xclang -ast-dump -fsyntax-only /tmp/test.cc

```
TranslationUnitDecl 0x2fde598
`-FunctionDecl 0x3018de8 <line:4:1> line:1:5 f 'int (int)'
 |- ParmVarDecl 0x3018d20  <col:7, col:11> col:11 used x 'int'
  `-CompoundStmt 0x3019060 <col:14, line:4:1>
    |-DeclStmt 0x3018ff0 <line:2:5, col:26>
    | `- VarDecl 0x3018ee8 <line:5, col:25> col:9 used result 'int' cinit
    |   `-ParenExpr 0x3018fd0 <col:18, col:25> 'int'
    |     `-BinaryOperator 0x3018fa8 <col:19, col:23> 'int' '/'
    |       |-ImplicitCastExpr 0x3018f90 <col:19> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x3018f48 <col:19> 'int' lvalue ParmVar
0x3018d20
    |       `-IntegerLiteral 0x3018f70 <col:23> 'int' 42
    `-ReturnStmt 0x3019048 <line:3:5, col:12>
      `-ImplicitCastExpr 0x3019030 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0x3019008 <col:12> 'int' lvalue Var 0x3018ee8
```

Clang: AST für ein einfaches Programm

```c
int f(int x) {
    int result = (x / 42);
    return result;
}
```
C

- Clang: C/C++-Frontend für LLVM
- Source-to-Source-Transformation
- `clang::VarDecl` ist ein `clang::NamedDecl`

$ clang -Xclang -ast-dump -fsyntax-only /tmp/test.cc

```
TranslationUnitDecl 0x2fde598
`-FunctionDecl                                  ne:4:1> line:1:5 f 'int (
 |- ParmVarDecl                                 ol:7, col:11> col:11 used
 `-CompoundSt                                   col:14, line:4:1>
   |-DeclStmt 0x3018ff0 <line:2:5, col:26>
   | `-VarDecl 0x3018ee8 <line:5, col:25> col:9 used result 'int' cinit
   |   `-ParenExpr 0x3018fd0 <col:18, col:25> 'int'
   |     `-BinaryOperator 0x                     l:23> 'int' '
   |       `-ImplicitCastExp                     > 'int' <LVa
   |         `-DeclRefExpr 0                     'int' lvalue P
0x3018d20
   |           `-IntegerLiteral 0x3018f70 <col:23> 'int' 42
   `-ReturnStmt 0x3019048 <line:3:5, col:12>
     `-ImplicitCastExpr 0x3019030 <col:12> 'int' <LValueToRValue>
       `-DeclRefExpr 0x3019008 <col:12> 'int' lvalue  Var  0x3018ee8
```

Verzeigerte Datenstruktur

Name der Deklaration

Knotentyp: C++-Klassen

Position innerhalb der Quelldatei

Typ der Deklaration

Clang: AST für ein einfaches Programm

```c
int f(int x) {
    int result = (x / 42);
    return result;
}
```
C

- Clang: C/C++-Frontend für LLVM
- Source-to-Source-Transformation
- `clang::VarDecl` ist ein `clang::NamedDecl`

$ clang -Xclang -ast-dump -fsyntax-only /tmp/test.cc

```
TranslationUnitDecl 0x2fde598
`-FunctionDecl 0x3018de8 <line:4:1> line:1:5 f 'int (int)'
 |- ParmVarDecl 0x3018d20  <col:7, col:11> col:11 used x 'int'
  `-CompoundStmt 0x3019060 <col:14, line:4:1>
    |-DeclStmt 0x3018ff0 <line:2:5, col:26>
    | `- VarDecl 0x3018ee8 <line:5, col:25> col:9 used result  'int'  cinit
    |   `-ParenExpr 0x3018fd0 <col:18, col:25> 'int'
    |     `-BinaryOperator 0x3018fa8 <col:19, col:23> 'int' '/'
    |       |-ImplicitCastExpr 0x3018f90 <col:19> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x3018f48 <col:19> 'int' lvalue ParmVar
0x3018d20
    |       `-IntegerLiteral 0x3018f70 <col:23> 'int' 42
    `-ReturnStmt 0x3019048 <line:3:5, col:12>
      `-ImplicitCastExpr 0x3019030 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0x3019008 <col:12> 'int' lvalue Var  0x3018ee8
```

Informationsfluss auf Bäumen

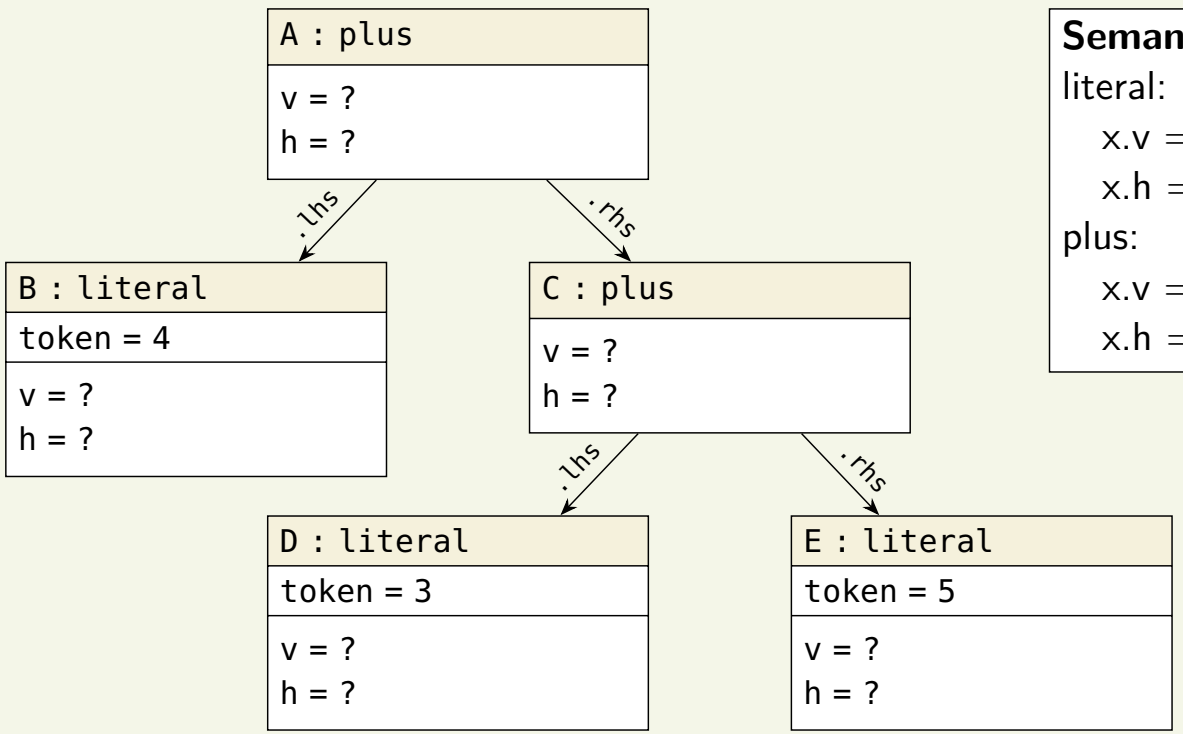
Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

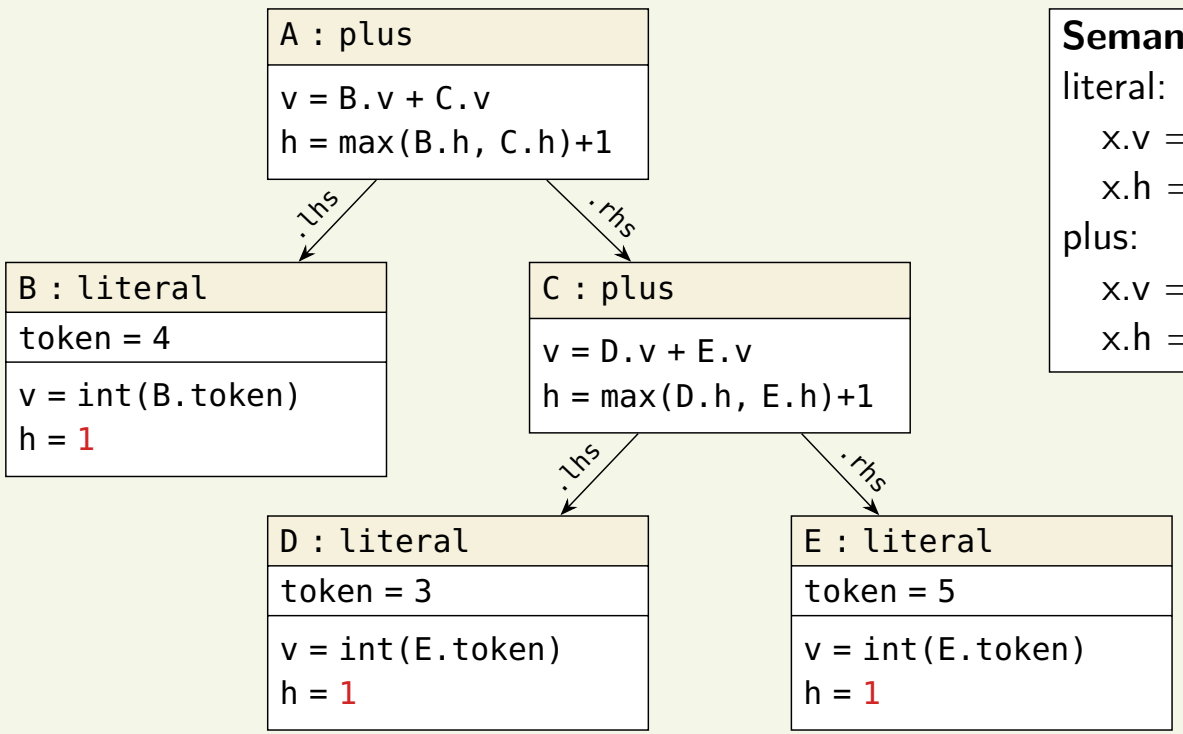
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

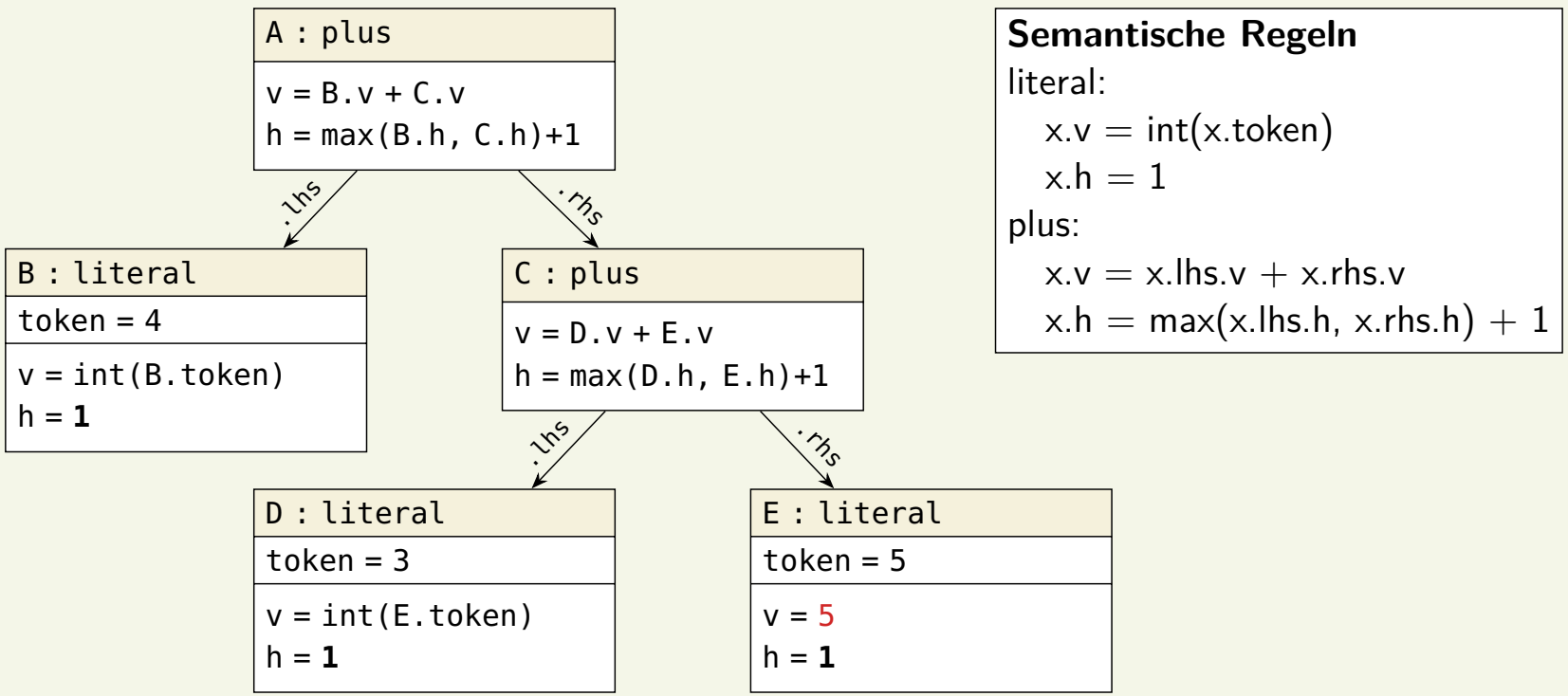
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

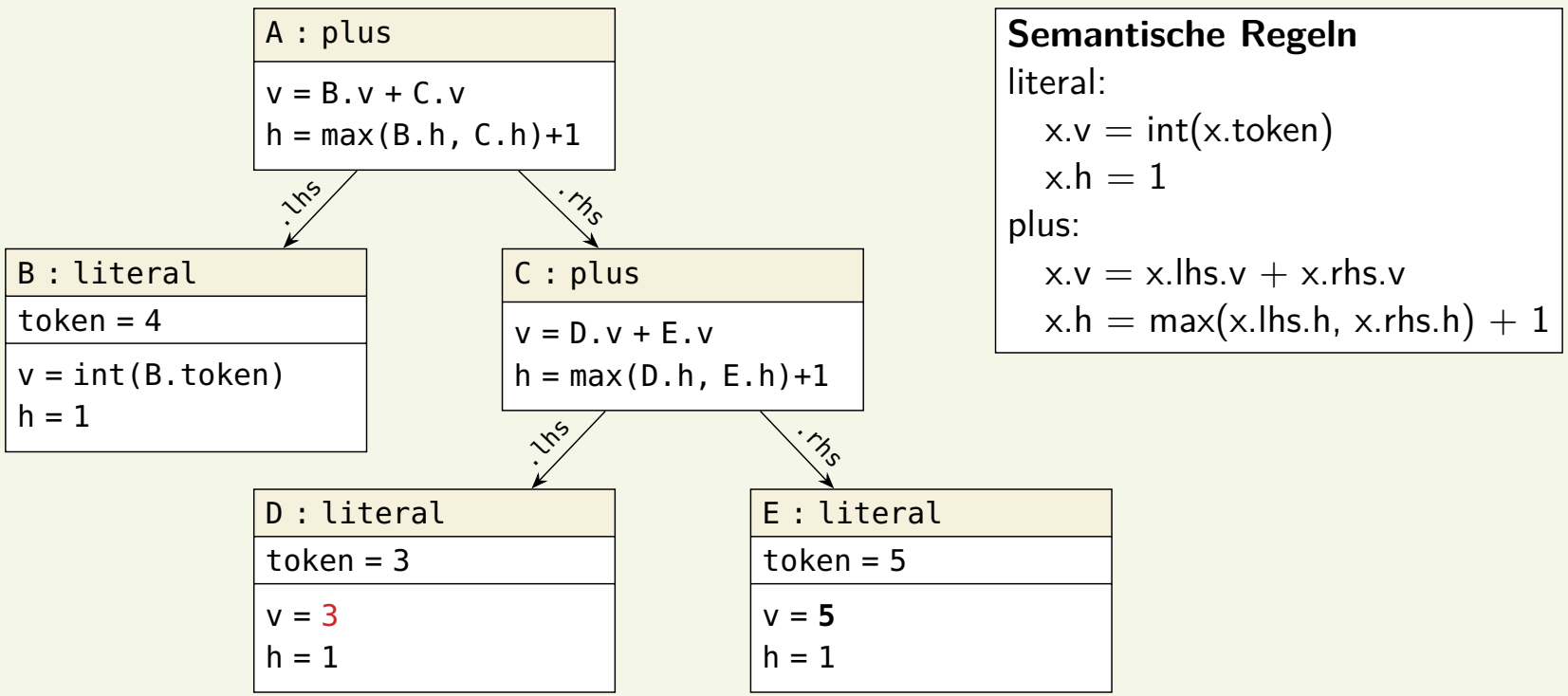
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

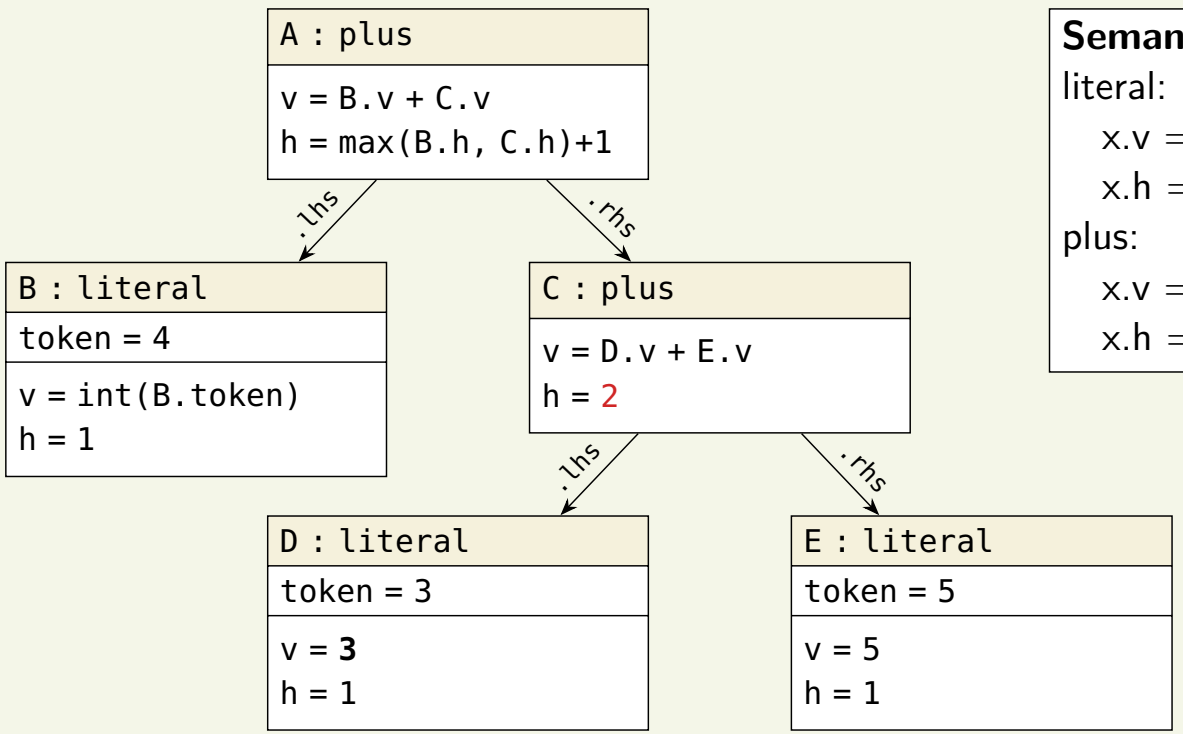
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

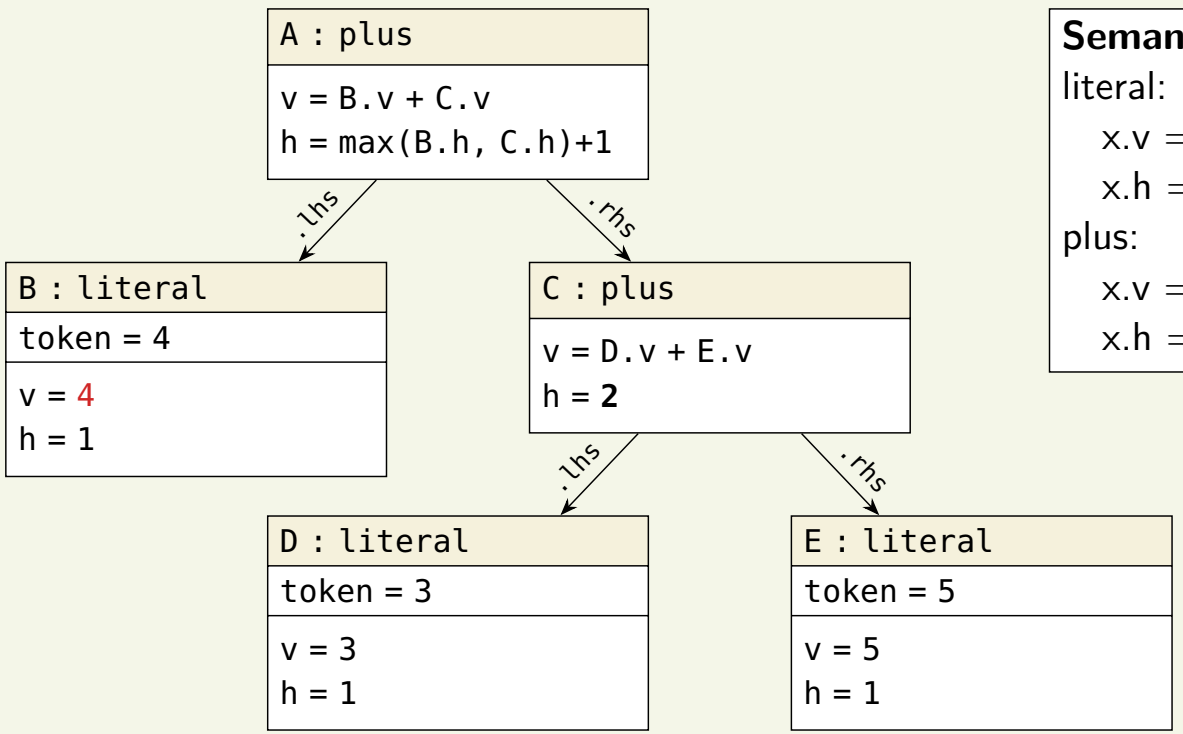
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

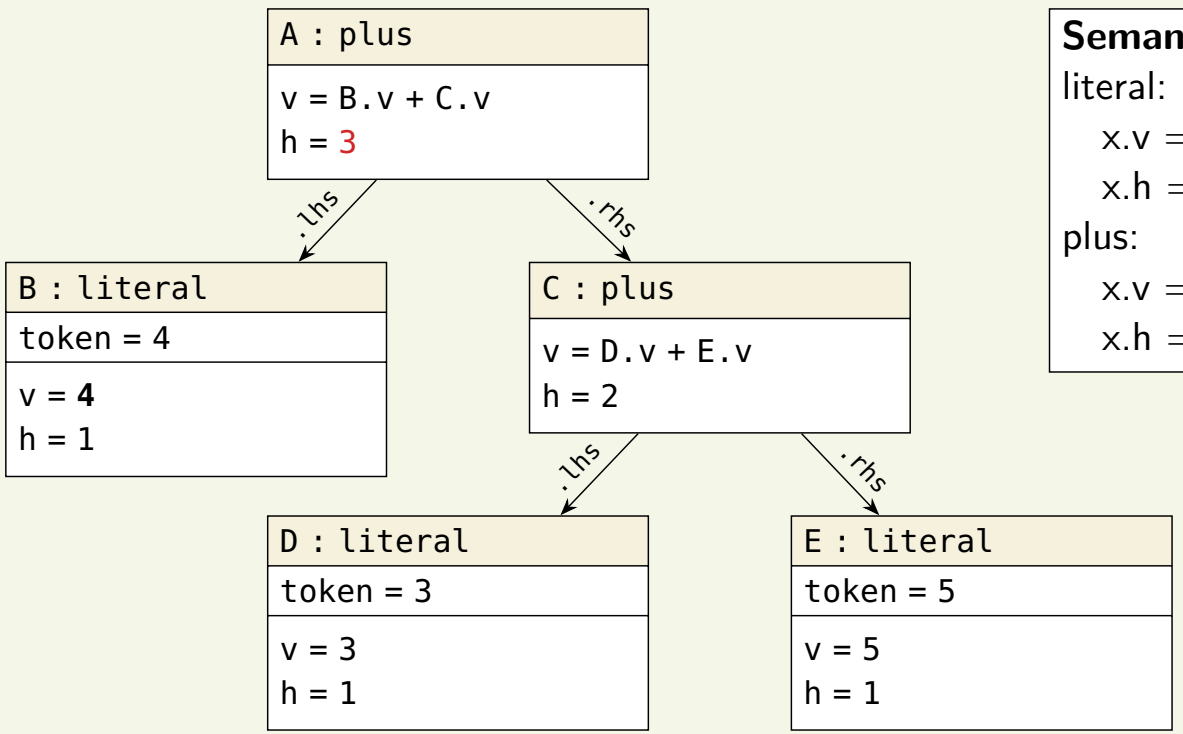
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

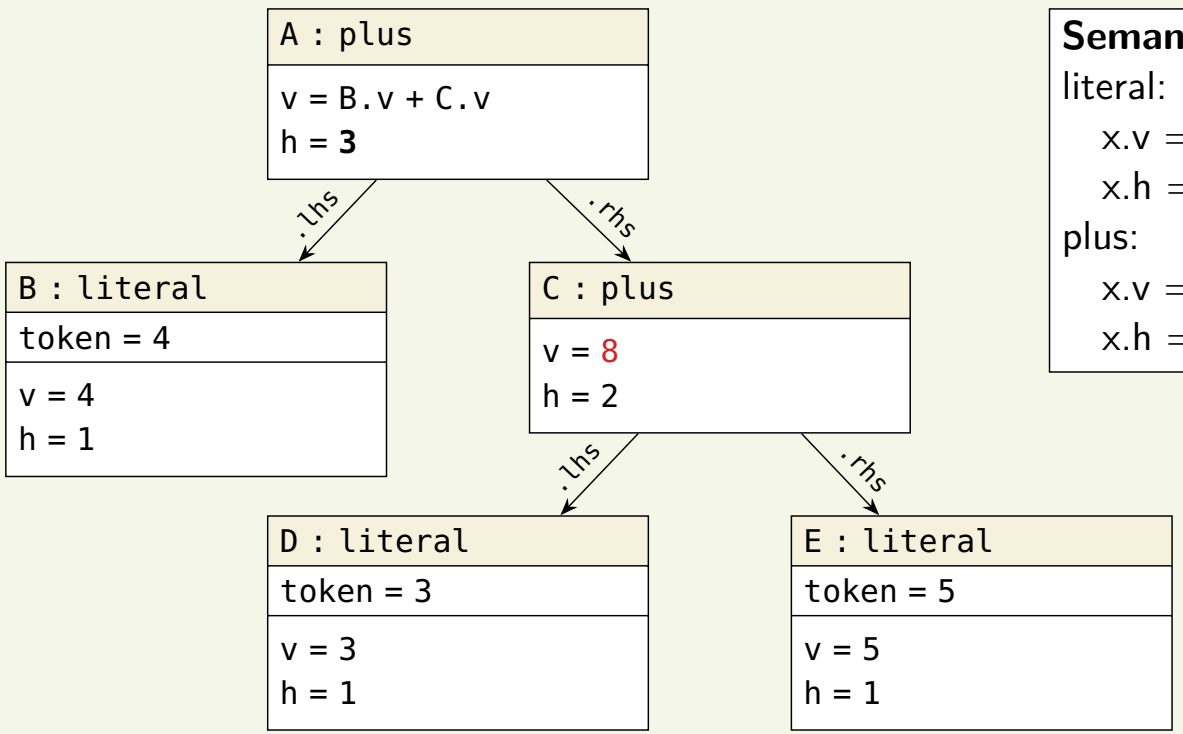
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

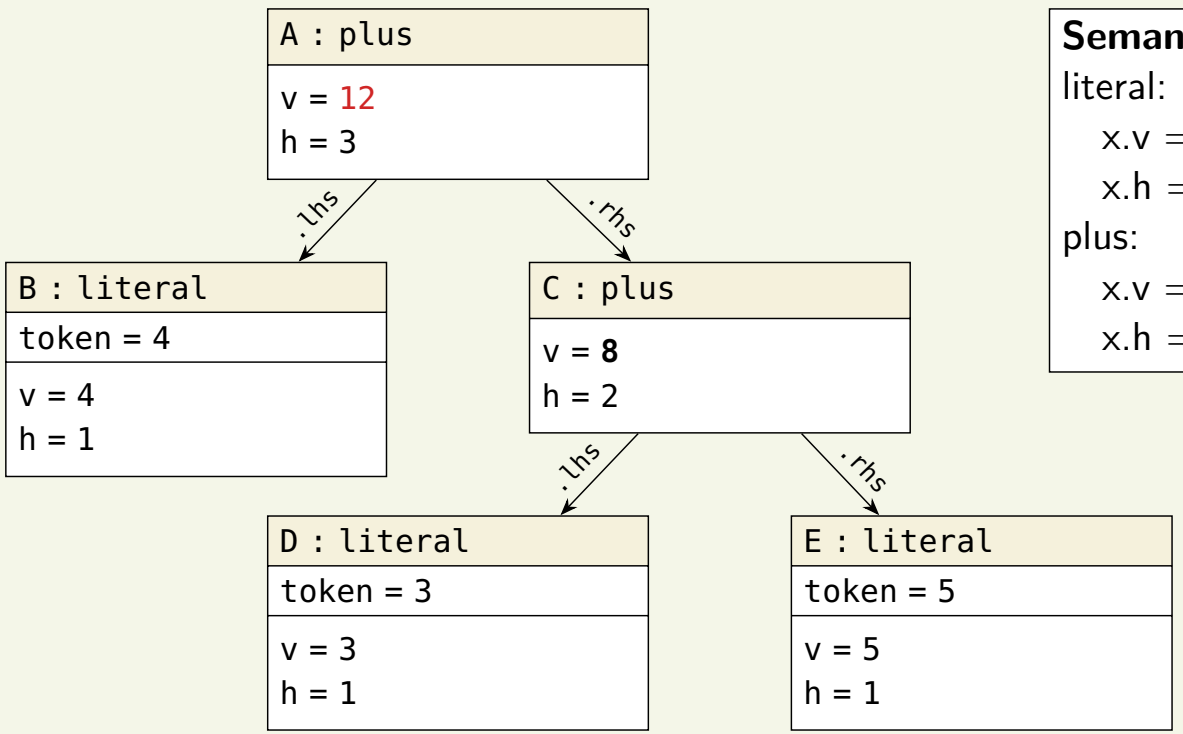
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



Semantische Regeln

literal:

$x.v = \text{int}(x.\text{token})$

$x.h = 1$

plus:

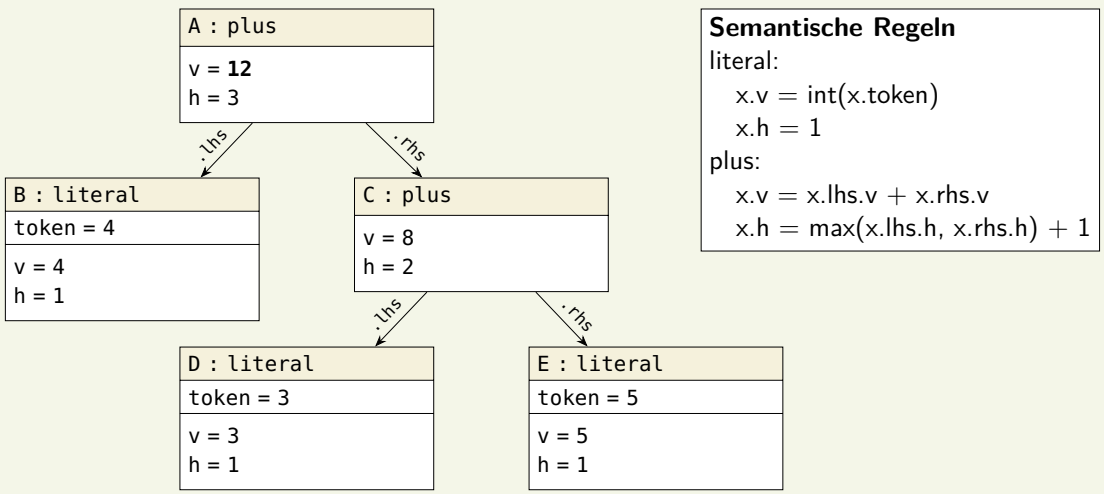
$x.v = x.\text{lhs}.v + x.\text{rhs}.v$

$x.h = \max(x.\text{lhs}.h, x.\text{rhs}.h) + 1$

Semantische Regeln als Attributgleichungen

- Die sem. Analyse berechnet die Werte der Attribute für jeden Knoten
  - Attribute hängen von anderen (benachbarten) AST-Knotenattributen ab.
  - Abstrakte Sicht: Gleichungssystem mit Gleichungen für jeden Knotentyp
  - Reihenfolge der Auswertung ist nicht festgelegt.

Beispiel: Höhe (h) und Wert (v) in einem Additionsbaum



- Attributgrammatiken annotieren die Regeln direkt an die Produktionen
  - `plus → expr + expr  { $0.h = max($1.h, $3.h) + 1 }`
  - Parseaktionen (Parsebaum→AST) sind bereits Attributgrammatiken

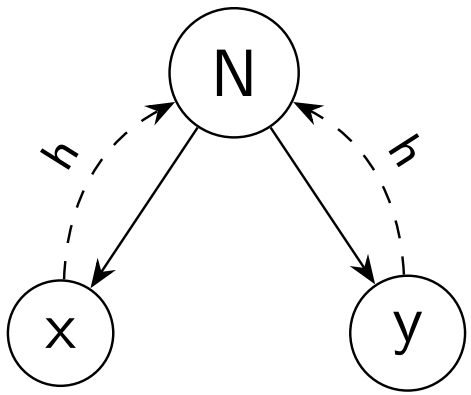
Abhängigkeitsgraphen und Attributtypen

Technische Universität Braunschweig

- Attribut-Gleichungssystem beinhaltet einen Abhängigkeitsgraphen
  - Für jedes Attribut auf der rechten Seite gibt es eine Abhängigkeit
  - Topologische Sortierung bestimmt die Auswertereihenfolge
  - Verschiedene Verfahren für unterschiedliche Typen von Abhängigkeiten

Abhängigkeitsgraphen und Attributtypen

- Attribut-Gleichungssystem beinhaltet einen Abhängigkeitsgraphen
  - Für jedes Attribut auf der rechten Seite gibt es eine Abhängigkeit
  - Topologische Sortierung bestimmt die Auswertereihenfolge
  - Verschiedene Verfahren für unterschiedliche Typen von Abhängigkeiten



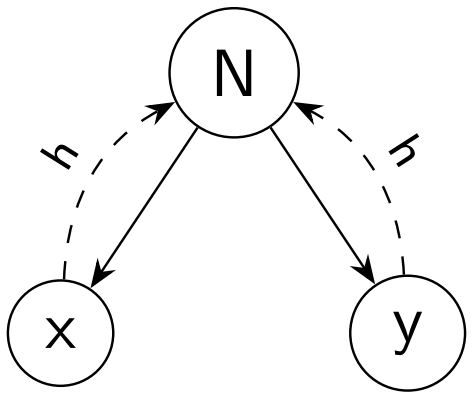
Synthetische Attribute

„von unten nach oben“

- S-Attribute
- Abhängig von Kindern
- Typen in Expressions
- Technik: Traversierung

Abhängigkeitsgraphen und Attributtypen

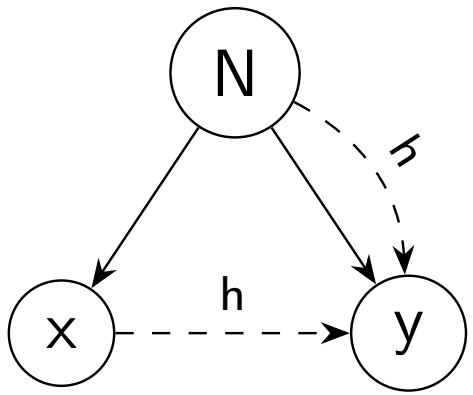
- Attribut-Gleichungssystem beinhaltet einen Abhängigkeitsgraphen
  - Für jedes Attribut auf der rechten Seite gibt es eine Abhängigkeit
  - Topologische Sortierung bestimmt die Auswertereihenfolge
  - Verschiedene Verfahren für unterschiedliche Typen von Abhängigkeiten



Synthetische Attribute

„von unten nach oben“

- S-Attribute
- Abhängig von Kindern
- Typen in Expressions
- Technik: Traversierung



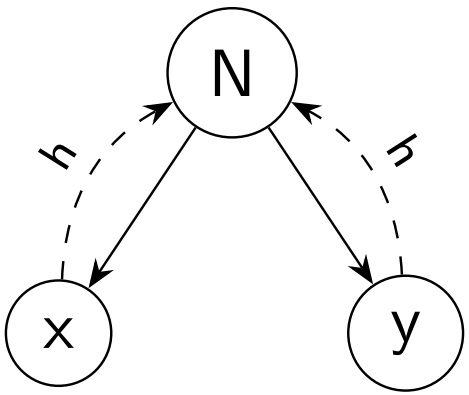
Geerbte Attribute

„von links nach rechts“

- L-Attribute
- Abhängig von Vorgängern und Nachbarn
- Namensauflösung
- Technik: Symboltabelle

Abhängigkeitsgraphen und Attributtypen

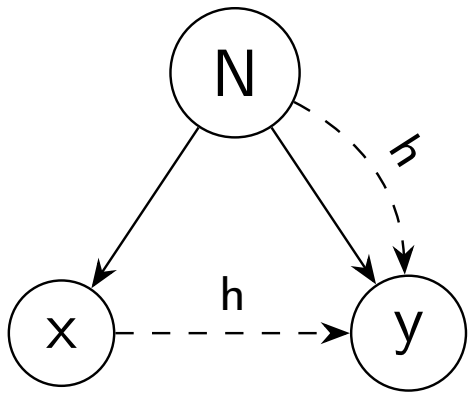
- Attribut-Gleichungssystem beinhaltet einen Abhängigkeitsgraphen
  - Für jedes Attribut auf der rechten Seite gibt es eine Abhängigkeit
  - Topologische Sortierung bestimmt die Auswertereihenfolge
  - Verschiedene Verfahren für unterschiedliche Typen von Abhängigkeiten



Synthetische Attribute

„von unten nach oben“

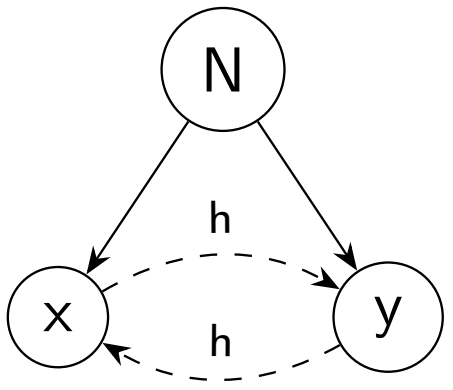
- S-Attribute
- Abhängig von Kindern
- Typen in Expressions
- Technik: Traversierung



Geerbte Attribute

„von links nach rechts“

- L-Attribute
- Abhängig von Vorgängern und Nachbarn
- Namensauflösung
- Technik: Symboltabelle



Zyklische Attribute

„all-over-the-place“

- Fixpunktberechnung
- Eher zu vermeiden
- Typinferenz in Haskell
- Technik: Unifikation

Berechnung der synthetischen Attribute

Wir müssen **jeden Knoten** besuchen, **knotenspezifischen** Code ausführen, und die Attributwerte **von unten nach oben** propagieren.

Berechnung der synthetischen Attribute

Wir müssen **jeden Knoten** besuchen, **knotenspezifischen** Code ausführen, und die Attributwerte **von unten nach oben** propagieren.

```python
def height(N):
  # Jedes Kind besuchen
  for child in N.children:
    height(child)
```

Python

Baumtraversierung (Post-Order)

- Bäume sind rekursiv.
- Jeden Knoten besuchen auch!

Berechnung der synthetischen Attribute

Wir müssen **jeden Knoten** besuchen, **knotenspezifischen** Code ausführen, und die Attributwerte **von unten nach oben** propagieren.

```python
def height(N):
  # Jedes Kind besuchen
  for child in N.children:
    height(child)

  # Knotenspezifischen Code
  if isinstance(N, literal):
    h = 1
  elif isinstance(N, plus):
    lhs, rhs = N.children
    h = max(lhs.h, rhs.h)+1
```

Python

Baumtraversierung (Post-Order)

- Bäume sind rekursiv.
- Jeden Knoten besuchen auch!

- Dynamischer Typ des Knotens
- Zugriff auf Attribute der Kinder
- Attributgleichung ausrechnen

Berechnung der synthetischen Attribute

Wir müssen **jeden Knoten** besuchen, **knotenspezifischen** Code ausführen, und die Attributwerte **von unten nach oben** propagieren.

```python
def height(N):
  # Jedes Kind besuchen
  for child in N.children:
    height(child)

  # Knotenspezifischen Code
  if isinstance(N, literal):
    h = 1
  elif isinstance(N, plus):
    lhs, rhs = N.children
    h = max(lhs.h, rhs.h)+1

  # Wert propagieren
  N.h = h
```

Python

Baumtraversierung (Post-Order)

- Bäume sind rekursiv.
- Jeden Knoten besuchen auch!

- Dynamischer Typ des Knotens
- Zugriff auf Attribute der Kinder
- Attributgleichung ausrechnen

- Setzen des Attributs
- Wert im Aufrufer verfügbar

Probleme von Ad-Hoc Traversierung

Ironie: Das geht aber noch mehr ad-hoc und unlesbar!

```
def height(N):
  if isinstance(N, literal):
    N.h = 1
  elif isinstance(N, plus):
    h = max(height(N.lhs),
            height(N.rhs))
    N.h = h + 1
  return N.h
```

Probleme von Ad-Hoc Traversierung

Ironie: Das geht aber noch mehr ad-hoc und unlesbar!

```
def height(N):
  if isinstance(N, literal):
    N.h = 1
  elif isinstance(N, plus):
    h = max(height(N.lhs),
            height(N.rhs))
    N.h = h + 1
  return N.h
```

- Die Attributgleichungen gehen im fragilen Boilerplate unter
  - Duplikation der Traversierungslogik für jedes Attribut
  - Duplikation der `isinstance` Kaskade für jedes Attribut
  - Ein neuer AST-Knotentyp $\rightarrow$ N Traversierungsfunktionen anpassen

Probleme von Ad-Hoc Traversierung

Technische Universität Braunschweig

Ironie: Das geht aber noch mehr ad-hoc und unlesbar!

```
def height(N):
  if isinstance(N, literal):
    N.h = 1
  elif isinstance(N, plus):
    h = max(height(N.lhs),
            height(N.rhs))
    N.h = h + 1
  return N.h
```

- Die Attributgleichungen gehen im fragilen Boilerplate unter
  - Duplikation der Traversierungslogik für jedes Attribut
  - Duplikation der `isinstance` Kaskade für jedes Attribut
  - Ein neuer AST-Knotentyp $\rightarrow$ N Traversierungsfunktionen anpassen
- Wende informatische Super-Power "Trennung der Belange" an!
  - Attributgleichungen werden in einer Besucherklasse gekapselt
  - Generischer Code für Traversierung in definierter Ordnung

Visitor-Pattern: Operationen auf Knoten kapseln

```C++
class Visitor { public:
    virtual void visit(literal& N) {
        N.h = 1;
    }
    virtual void visit(plus& N) {
        N.h = max(N.lhs.h, N.rhs.h) + 1;
    }
};
```

- Bündelung der Attributgleichungen in einem Visitor-Objekt
  - Die `visit()` -Funktion wird für jeden Knotentyp überladen.
  - Visitor-Objekt kann zusätzlichen Zustand als Member speichern
  - Vererbungshierarchie kann beachtet werden (`void visit(NamedDecl D)`)

Visitor-Pattern: Operationen auf Knoten kapseln

```C++
class Visitor { public:
    virtual void visit(literal& N) {
        N.h = 1;
    }
    virtual void visit(plus& N) {
        N.h = max(N.lhs.h, N.rhs.h) + 1;
    }
};
```

- Bündelung der Attributgleichungen in einem Visitor-Objekt
  - Die `visit()` -Funktion wird für jeden Knotentyp überladen.
  - Visitor-Objekt kann zusätzlichen Zustand als Member speichern
  - Vererbungshierarchie kann beachtet werden (`void visit(NamedDecl D)`)

```
void
traversal(Visitor& v, Tree& t) {
  for (Tree& c : t.children()) {
    traversal(v, c);
  }
  v.visit(t);
}
```

Generische Traversierung

- Knoten haben Liste der Kinder
- Definierte Besuchsreihenfolge
- Visitor wird auf jeden Knoten angewendet

Visitor-Pattern: Operationen auf Knoten kapseln

```C++
class Visitor { public:
    virtual void visit(literal& N) {
        N.h = 1;
    }
    virtual void visit(plus& N) {
        N.h = max(N.lhs.h, N.rhs.h) + 1;
    }
};
```

- Bündelung der Attributgleichungen in einem Visitor-Objekt
  - Die `visit()` -Funktion wird für jeden Knotentyp überladen.
  - Visitor-Objekt kann zusätzlichen Zustand als Member speichern
  - Vererbungshierarchie kann beachtet werden (`void visit(NamedDecl D)`)

```
void
traversal(Visitor& v, Tree& t) {
  for (Tree& c : t.children()) {
    traversal(v, c);
  }
  v.visit(t);
}
```

Generische Traversierung

- Knoten haben Liste der Kinder

Das ist so kaputt!
Dynamischer Dispatch nur im nullten Argument!
Ruft immer `Visitor::visit(Tree)` auf!

Visitor-Pattern: Double Dynamic Dispatch

Statische Typen an der Aufrufstelle: `visit(Visitor, Tree)`

Bei der Traversierung soll `visit(v, t)` für das konkrete Visitor-Objekt und spezifisch für den AST-Knoten aufgerufen werden.

```
visit(HeightVisitor, literal)      visit(ValueVisitor, literal)
visit(HeightVisitor, plus)         visit(ValueVisitor, plus)
```

Visitor-Pattern: Double Dynamic Dispatch

Statische Typen an der Aufrufstelle: `visit(Visitor, Tree)`

Bei der Traversierung soll `visit(v, t)` für das konkrete Visitor-Objekt und spezifisch für den AST-Knoten aufgerufen werden.

```
visit(HeightVisitor, literal)      visit(ValueVisitor, literal)
visit(HeightVisitor, plus)         visit(ValueVisitor, plus)
```

- Dynamischer Dispatch in den **ersten zwei Argumenten** nötig
  - Emulation mittels zweifachem virtuellen Methodenaufruf.

```
void traversal(Visitor& v, Tree& t) {
  ...
  t.accept(v);
  ...
}
```

C++

Dynamic Dispatch

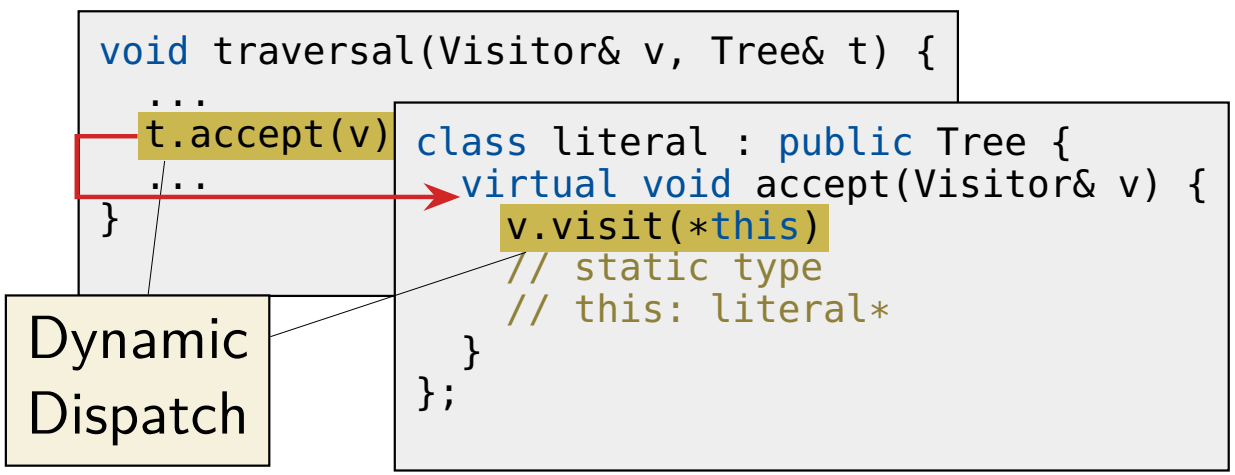
Visitor-Pattern: Double Dynamic Dispatch

Statische Typen an der Aufrufstelle: `visit(Visitor, Tree)`

Bei der Traversierung soll `visit(v, t)` für das konkrete Visitor-Objekt und spezifisch für den AST-Knoten aufgerufen werden.

```
visit(HeightVisitor, literal)      visit(ValueVisitor, literal)
visit(HeightVisitor, plus)         visit(ValueVisitor, plus)
```

- Dynamischer Dispatch in den **ersten zwei Argumenten** nötig
  - Emulation mittels zweifachem virtuellen Methodenaufruf.



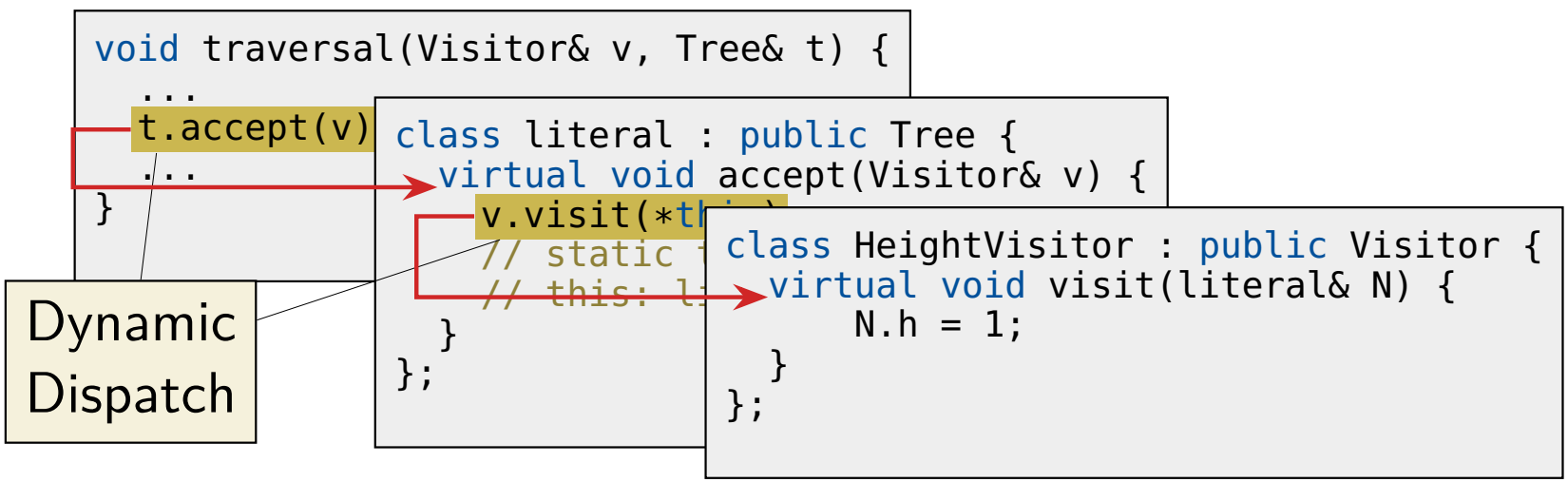
Visitor-Pattern: Double Dynamic Dispatch

Statische Typen an der Aufrufstelle: `visit(Visitor, Tree)`

Bei der Traversierung soll `visit(v, t)` für das konkrete Visitor-Objekt und spezifisch für den AST-Knoten aufgerufen werden.

```
visit(HeightVisitor, literal)      visit(ValueVisitor, literal)
visit(HeightVisitor, plus)         visit(ValueVisitor, plus)
```

- Dynamischer Dispatch in den **ersten zwei Argumenten** nötig
  - Emulation mittels zweifachem virtuellen Methodenaufruf.



Visitor-Pattern: Dispatch mit Introspection

Technische Universität Braunschweig

- Python hat Überladung nur im ersten Argument (self), dafür dynamisch!
  - Keine statischen Typen $\Rightarrow$ Keine doppelte Definition von Methoden.
  - Jedoch kann man Namensauflösung zur Laufzeit durchführen.

Visitor-Pattern: Dispatch mit Introspection

- Python hat Überladung nur im ersten Argument (self), dafür dynamisch!
  - Keine statischen Typen ⇒ Keine doppelte Definition von Methoden.
  - Jedoch kann man Namensauflösung zur Laufzeit durchführen.

```
def traversal(V, T):
  # Depth-First Order
  for child in T.children:
    traversal(V, child)

  # Introspection
  T_name = type(T).__qualname__
  M_name = "visit_" + T_name
  method = getattr(V, M_name)

  # Aufruf des Visitors
  method(T)
```

```
class HeightVisitor:
  def visit_literal(self, N):
    N.h = 1

  def visit_plus(self, N):
    h = max(N.lhs.h,
            N.rhs.h)
    N.h = h + 1
```

```
visitor = HeightVisitor()
traversal(visitor, tree)
```

- Emulation von Überladung durch Introspektion
  - `type(T)` extrahiert den **dynamischen** Typen des Knotens
  - `M_name = "visit_"+ ...` erzeugt dynamisch einen **Namen**
  - `getattr(V, "visit_plus")` löst den Namen; gibt die Methode zurück

Berechnung des Typen und Typkonsistenz

Technische Universität Braunschweig

- S-Attribut: Für Expressions berechnen wir Typen (bottom-up)
  - Blätter: Konstante Literale (23, ”foobar”) haben feste, eingebaute Typen
    Namensreferenzen haben Typen der Deklaration (später mehr)
  - Innere Knoten: Jeder Knotentyp hat eine Typ-Gleichung.
    Jeder Knoten propagiert seinen Ergebnistypen nach oben.

Typberechnung und -prüfung geschieht „in einem Rutsch“!

Berechnung des Typen und Typkonsistenz

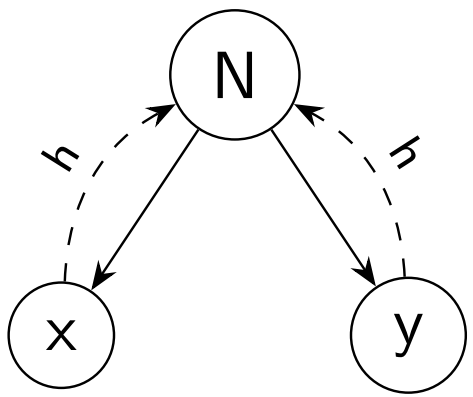
- S-Attribut: Für Expressions berechnen wir Typen (bottom-up)
  - Blätter: Konstante Literale (23, ”foobar”) haben feste, eingebaute Typen
    Namensreferenzen haben Typen der Deklaration (später mehr)
  - Innere Knoten: Jeder Knotentyp hat eine Typ-Gleichung.
    Jeder Knoten propagiert seinen Ergebnistypen nach oben.

Typberechnung und -prüfung geschieht „in einem Rutsch“!

```
class TypeVisitor:
  def visit_literal(self, T):
    # Ganzzahl-Literal
    T.Type = int
```

Minimalbeispiel

- Monomorphes Typsystem
- Ganzzahl und Pointertyp
- Addition nur auf Ganzzahlen



Berechnung des Typen und Typkonsistenz

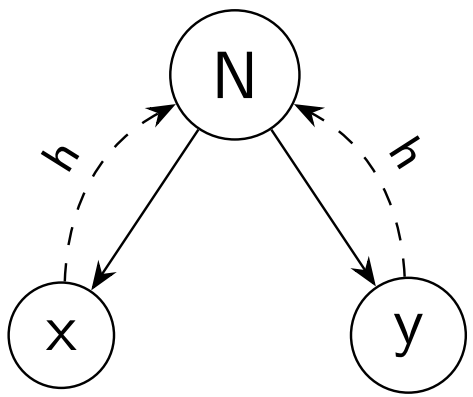
- S-Attribut: Für Expressions berechnen wir Typen (bottom-up)
  - Blätter: Konstante Literale (23, ”foobar”) haben feste, eingebaute Typen
    Namensreferenzen haben Typen der Deklaration (später mehr)
  - Innere Knoten: Jeder Knotentyp hat eine Typ-Gleichung.
    Jeder Knoten propagiert seinen Ergebnistypen nach oben.

Typberechnung und -prüfung geschieht „in einem Rutsch“!

```
class TypeVisitor:
  def visit_literal(self, T):
    # Ganzzahl-Literal
    T.Type = int
  def visit_addressof(self, T):
    # Typausdruck: &(op)
    T.Type = pointer(T.op.Type)
```

Minimalbeispiel

- Monomorphes Typsystem
- Ganzzahl und Pointertyp
- Addition nur auf Ganzzahlen



Berechnung des Typen und Typkonsistenz

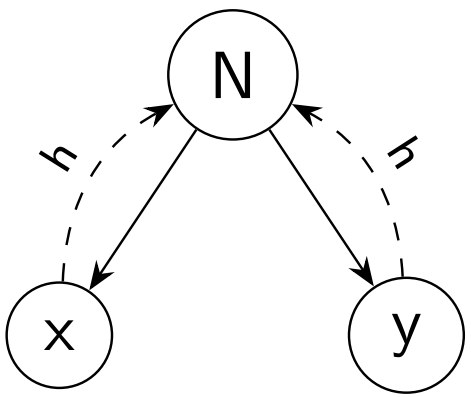
- S-Attribut: Für Expressions berechnen wir Typen (bottom-up)
  - Blätter: Konstante Literale (23, "foobar") haben feste, eingebaute Typen
    Namensreferenzen haben Typen der Deklaration (später mehr)
  - Innere Knoten: Jeder Knotentyp hat eine Typ-Gleichung.
    Jeder Knoten propagiert seinen Ergebnistypen nach oben.

Typberechnung und -prüfung geschieht „in einem Rutsch"!

```
class TypeVisitor:
  def visit_literal(self, T):
    # Ganzzahl-Literal
    T.Type = int
  def visit_addressof(self, T):
    # Typausdruck: &(op)
    T.Type = pointer(T.op.Type)
  def visit_add(self, T):
    L, R = T.lhs.Type, T.rhs.Type
    # Konsistenzprüfung
    if not L.equal(int) \
       or not L.equal(R):
      self.error(...)
    # Typausdruck: lhs + rhs
    T.Type = L
```

Minimalbeispiel

- Monomorphes Typsystem
- Ganzzahl und Pointertyp
- Addition nur auf Ganzzahlen

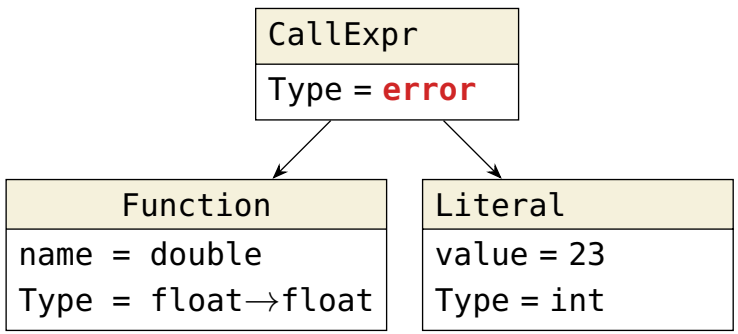


Coercion: Implizite Typumwandlung

Automatisch angewendete implizite Typumwandlung

Was passiert bei inkompatiblen Typen?

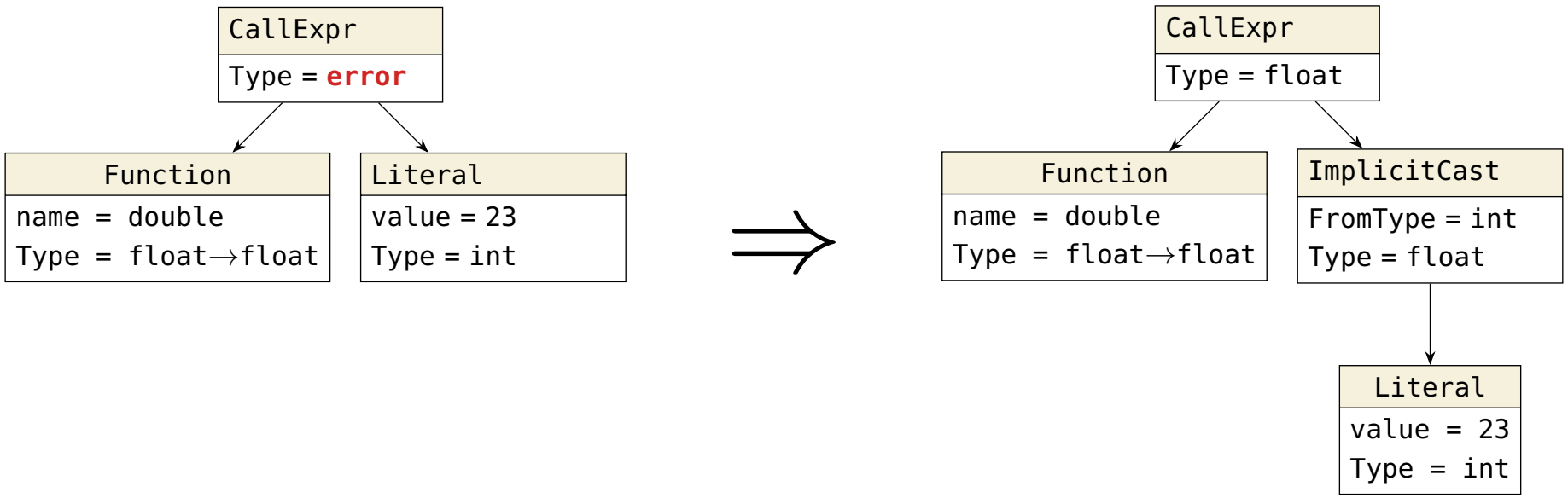
Passen geforderter (z.B. Parameter) und vorhandener Typ (z.B. Argument) nicht zusammen, kann eine **implizite** Typumwandlung eingefügt werden.



Automatisch angewendete implizite Typumwandlung

Was passiert bei inkompatiblen Typen?

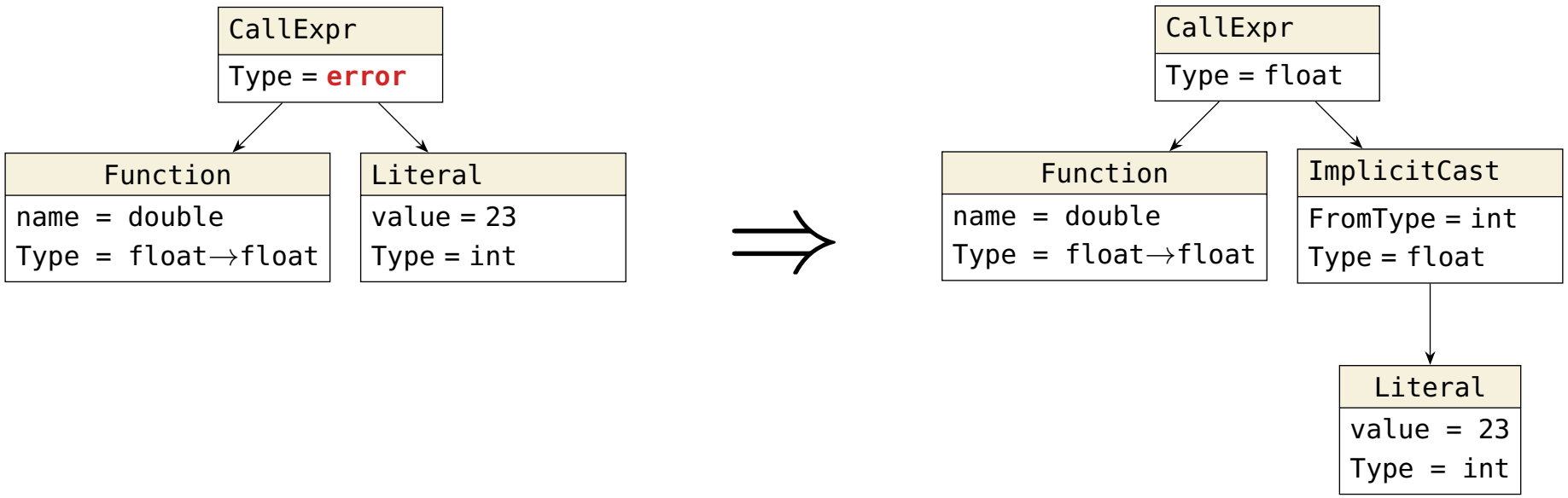
Passen geforderter (z.B. Parameter) und vorhandener Typ (z.B. Argument) nicht zusammen, kann eine **implizite** Typumwandlung eingefügt werden.



Automatisch angewendete implizite Typumwandlung

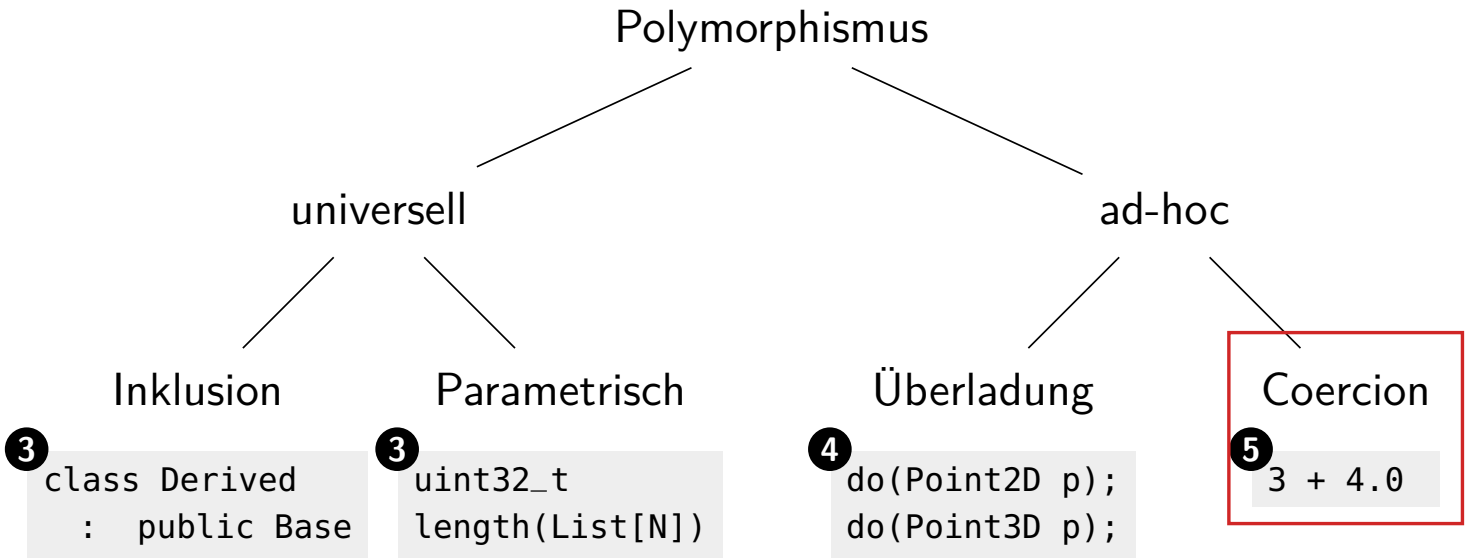
Was passiert bei inkompatiblen Typen?

Passen geforderter (z.B. Parameter) und vorhandener Typ (z.B. Argument) nicht zusammen, kann eine **implizite** Typumwandlung eingefügt werden.



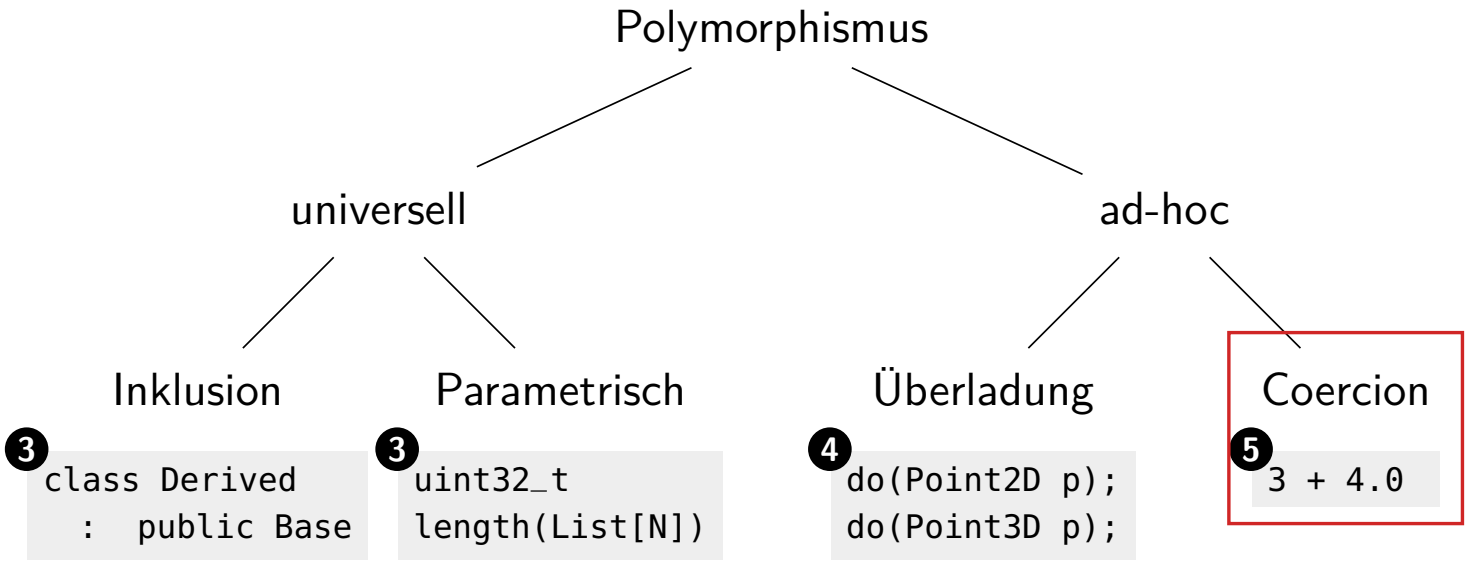
- Sprachen definieren eine Reihe von impliziten Typumwandlungen
  - Ganzzahlweitung: Umwandlung in breitere Integer ohne Informationsverlust
  - C++: `char` → `short` → `int` → `long`
  - C: Implizite Umwandlung von `void*` zu beliebigen Zeigern (z.B. `int *`)

Einordnung der automatischen Typumwandlung



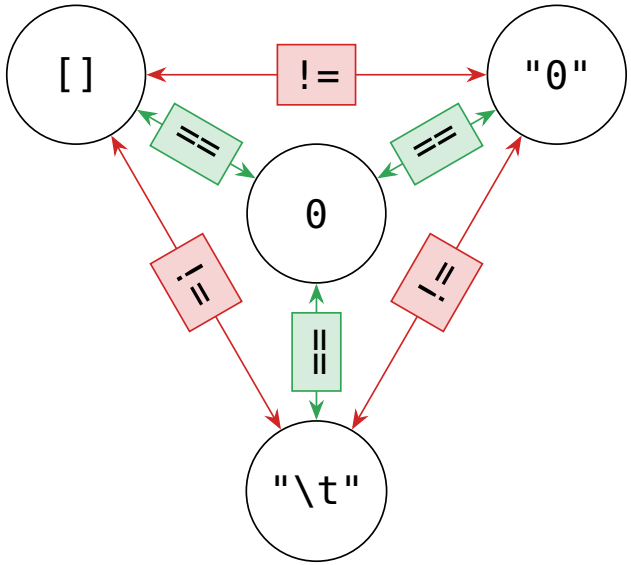
- Automatisch umgewandelte Typen verhalten sich polymorph!
  - Ad-Hoc: Die Umwandlung ist abhängig vom konkreten Kontext
  - Typumwandlung ist polymorph auf Seite des Produzenten
    Überladung ist polymorph auf Seite des Konsumenten

Einordnung der automatischen Typumwandlung



- Automatisch umgewandelte Typen verhalten sich polymorph!
  - Ad-Hoc: Die Umwandlung ist abhängig vom konkreten Kontext
  - Typumwandlung ist polymorph auf Seite des Produzenten
    Überladung ist polymorph auf Seite des Konsumenten
- Manche Sprachen erlauben nutzerdefinierte implizite Umwandlungen.
  - Converting constructors (C++): Konstruktoren mit genau einem Argument ergänzen die eingebauten Regeln zur Typumwandlung.

Probleme der impliziten Typumwandlung



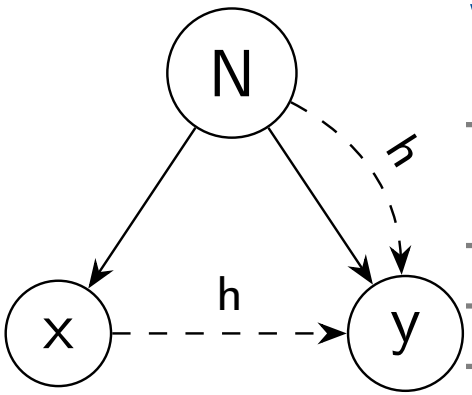
- Implizite Typumwandlung können leicht zu Verwirrung führen!
  - Javascript: Umwandlungen von Strings und leeren Arrays zur Zahl 0
    „When using two equals signs [...] some funky conversions take place“
  - Inkonsistente Gleichheit: `([]!="0")` vs. `(([]==0)&&(0=="0"))`

⇒ Die traurige Wahrheit: JavaScript hat zusätzlich einen ===-Operator.

Namensauflösung mittels Symboltabelle

Statische Namensauflösung als L-Attribut

- Referenzen beziehen sich auf benannte aber entfernte Deklarationen
  - Deklaration muss vor der Benutzung stattfinden (im AST links)
  - Welche Deklaration gemeint ist, ist ein L-Attribut.

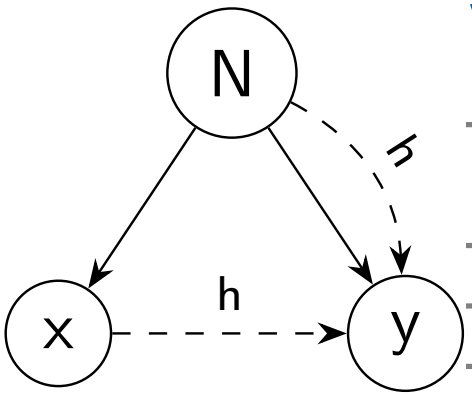


Vorgehen der Namensauflösung

- Traversierung aller Deklarationen, Referenzen und Namensraumkonstrukte
- Symboltabelle speichert die aktuell sichtbaren Namen
- Fortwährende Aktualisierung der Tabelle
- Auflösung von Referenzen über API der Symboltabelle

Statische Namensauflösung als L-Attribut

- Referenzen beziehen sich auf benannte aber entfernte Deklarationen
  - Deklaration muss vor der Benutzung stattfinden (im AST links)
  - Welche Deklaration gemeint ist, ist ein L-Attribut.



Vorgehen der Namensauflösung

- Traversierung aller Deklarationen, Referenzen und Namensraumkonstrukte
- Symboltabelle speichert die aktuell sichtbaren Namen
- Fortwährende Aktualisierung der Tabelle
- Auflösung von Referenzen über API der Symboltabelle

- Symboltabelle: Mögliche API in Java

```
interface SymbolTable {
  ...
  public void openNamespace(Tree NS);
  public void createName(Name name, Tree node);
  public Tree findName(Name name);
  public void closeNamespace(Tree NS);
  ...
}
```

Java

AST-Traversierung mit Symboltabelle

```
class NameVisitor:
  def __init__(self):
    self.ST = SymbolTable()
```

- Visitor hält eine Symboltabelle, die den aktuellen Kontext speichert.
- `self.ST` ist der „Datenrucksack“ für die Traversierung.

AST-Traversierung mit Symboltabelle

```
class NameVisitor:
  def __init__(self):
    self.ST = SymbolTable()
```

```
  def pre_NamespaceDecl(self, NS):
    self.ST.openNamespace(NS)

  def post_NamespaceDecl(self, NS):
    self.ST.closeNamespace(NS)
```

- Visitor hält eine Symboltabelle, die den aktuellen Kontext speichert.
- `self.ST` ist der „Datenrucksack“ für die Traversierung.
- Visitor muss es erlauben Code vor (`pre_`) und nach (`post_`) Besuch der Kindknoten auszuführen.
- Einige AST-Knoten öffnen einen neuen Namensraum.

```
def traversal(V, T):
    # V.pre_<DynamicTypeName>(T)
    dispatch(V, "pre_", T)

    for child in T.children:
        traversal(V, child)

    # V.post_<DynamicTypeName>(T)
    dispatch(V, "post_", T)
```

AST-Traversierung mit Symboltabelle

Technische Universität Braunschweig

```
class NameVisitor:
  def __init__(self):
    self.ST = SymbolTable()
```

```
  def pre_NamespaceDecl(self, NS):
    self.ST.openNamespace(NS)

  def post_NamespaceDecl(self, NS):
    self.ST.closeNamespace(NS)
```

```
  def pre_VarDecl(self, D):
    N = D.identifier
    self.ST.createName(N, D)
```

- Visitor hält eine Symboltabelle, die den aktuellen Kontext speichert.
- `self.ST` ist der „Datenrucksack“ für die Traversierung.
- Visitor muss es erlauben Code vor (`pre_`) und nach (`post_`) Besuch der Kindknoten auszuführen.
- Einige AST-Knoten öffnen einen neuen Namensraum.
- Deklarationen führen Namen (`N`) für AST-Knoten (`fn`) ein

AST-Traversierung mit Symboltabelle

Technische Universität Braunschweig

```
class NameVisitor:
  def __init__(self):
    self.ST = SymbolTable()
```

- Visitor hält eine Symboltabelle, die den aktuellen Kontext speichert.
- `self.ST` ist der „Datenrucksack“ für die Traversierung.

```
  def pre_NamespaceDecl(self, NS):
    self.ST.openNamespace(NS)

  def post_NamespaceDecl(self, NS):
    self.ST.closeNamespace(NS)
```

- Visitor muss es erlauben Code vor (`pre_`) und nach (`post_`) Besuch der Kindknoten auszuführen.
- Einige AST-Knoten öffnen einen neuen Namensraum.

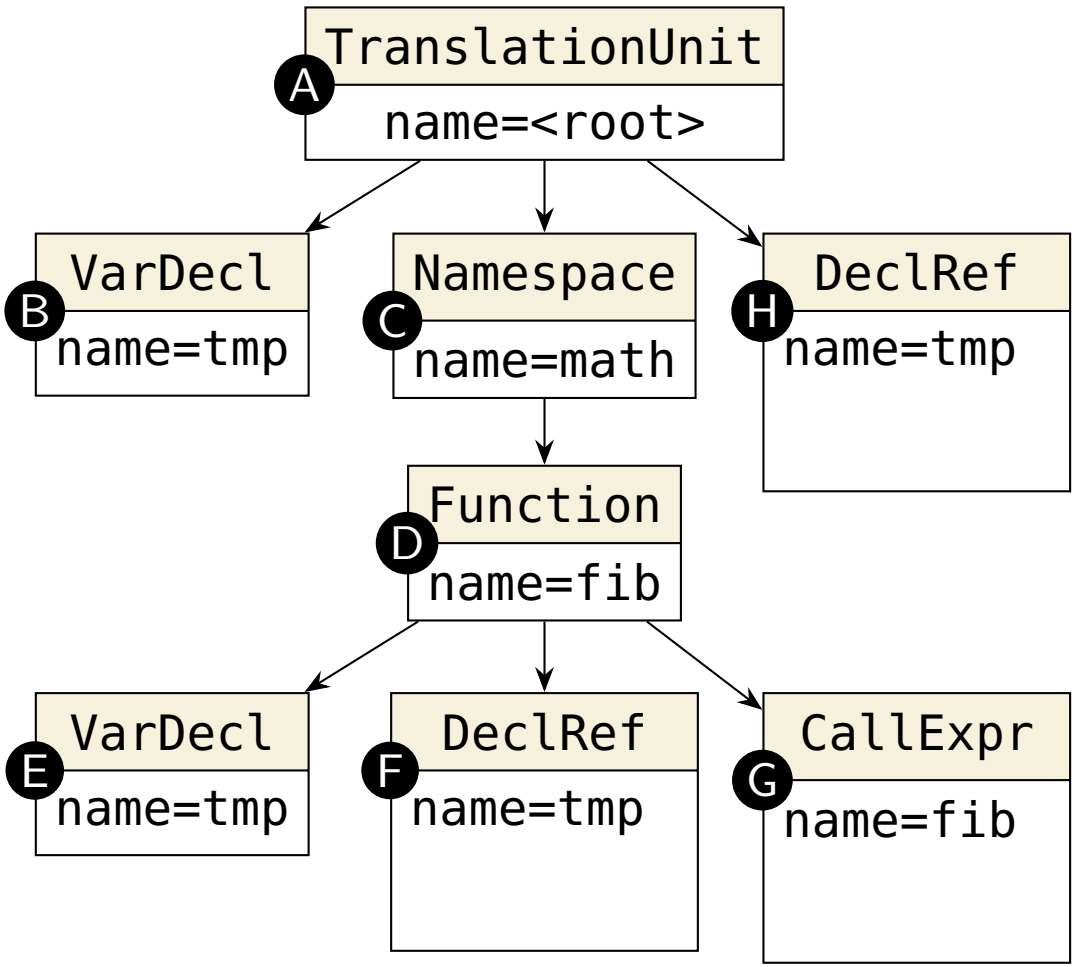
```
  def pre_VarDecl(self, D):
    N = D.identifier
    self.ST.createName(N, D)
```

- Deklarationen führen Namen (`N`) für AST-Knoten (`fn`) ein

```
  def pre_DeclRefExpr(self, R):
    N = R.identifier
    decl = self.ST.findName(N)
    R.decl = decl
```

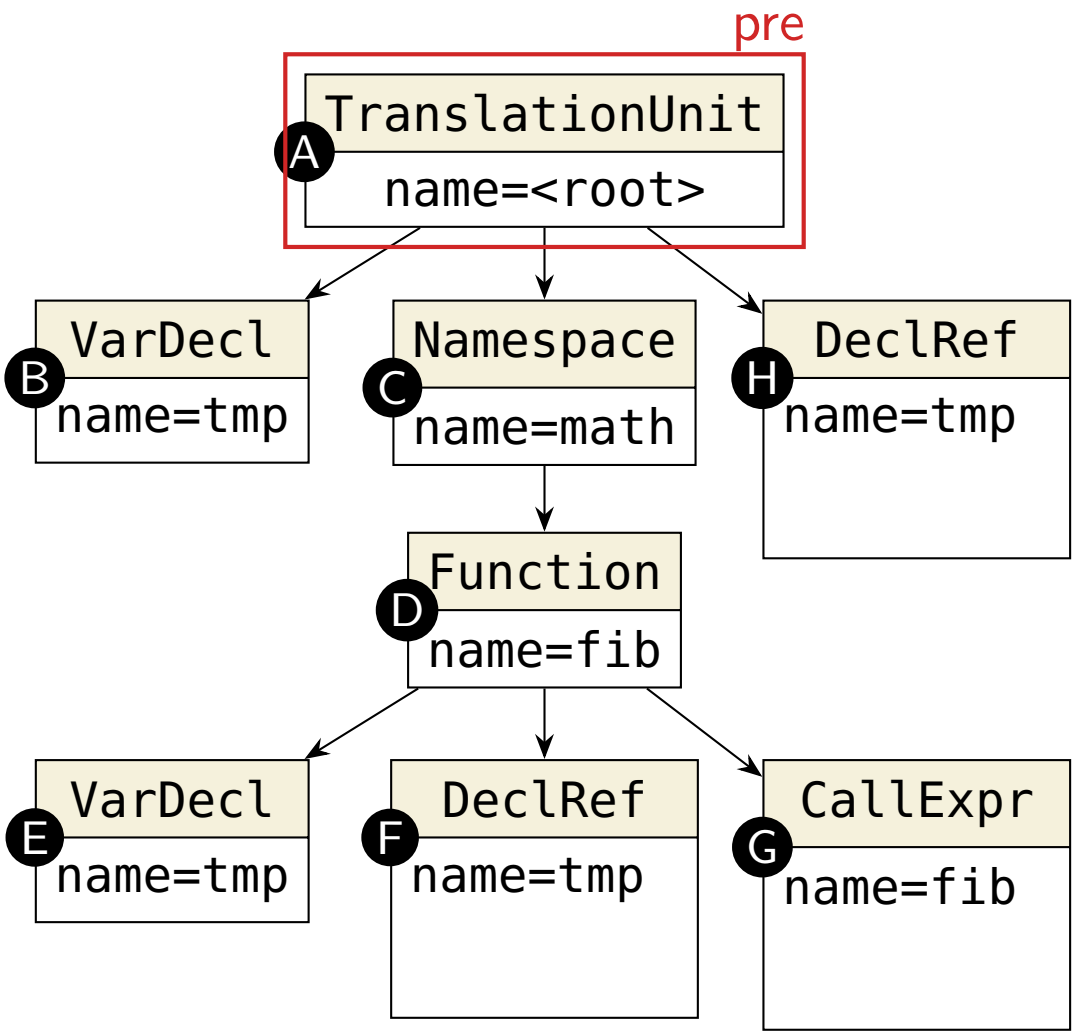
- Referenz `R` mittels Tabelle auflösen
- Ergebnis (`decl`) als Attribut speichern

Symboltabelle als Stack von Namensräumen



Symbol Table

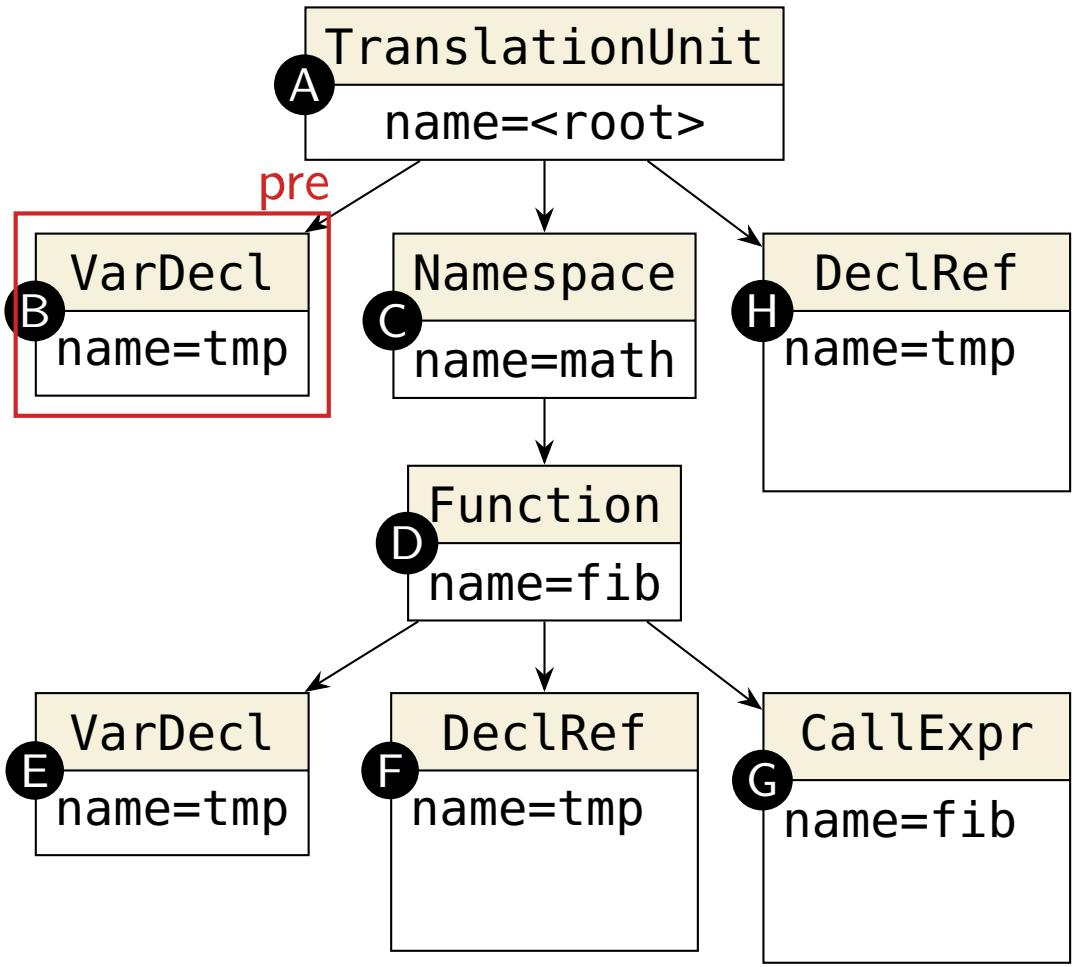
Symboltabelle als Stack von Namensräumen



Symbol Table

Namespace:	A

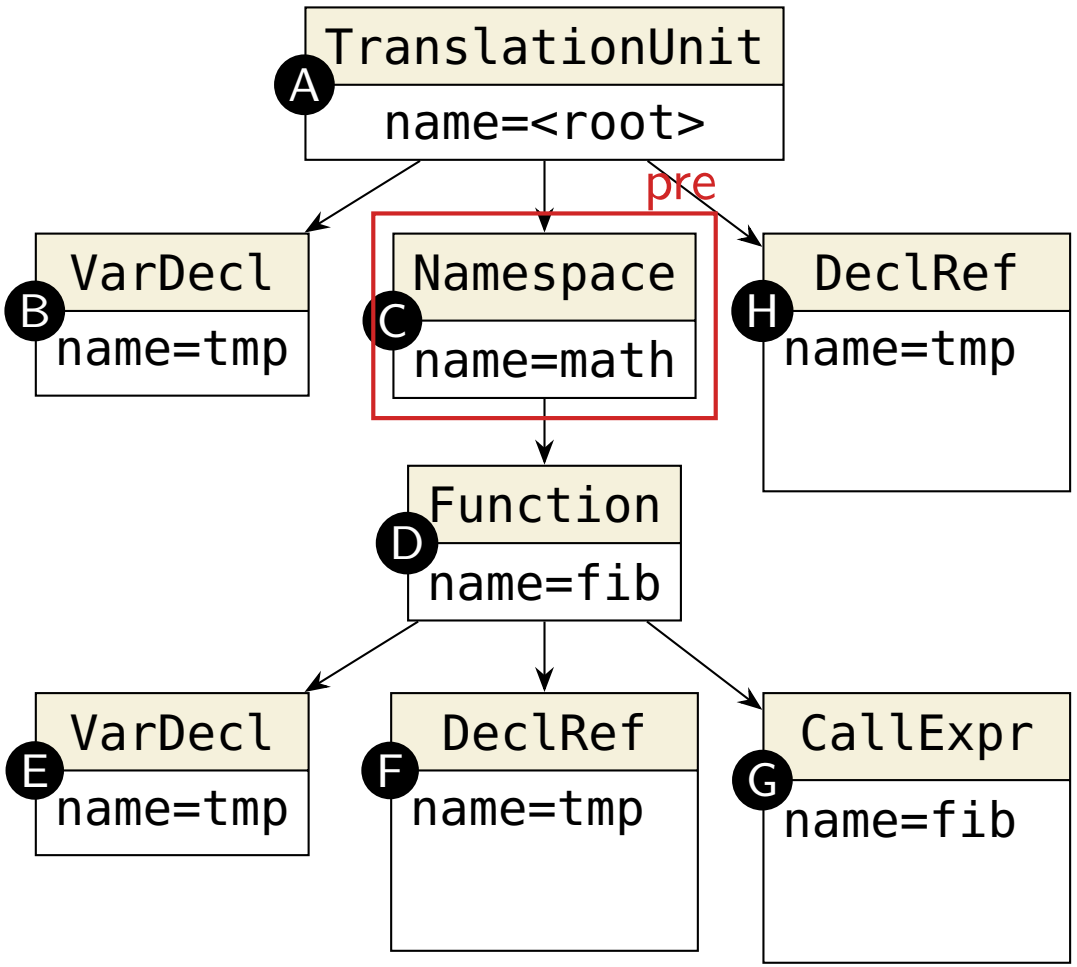
Symboltabelle als Stack von Namensräumen



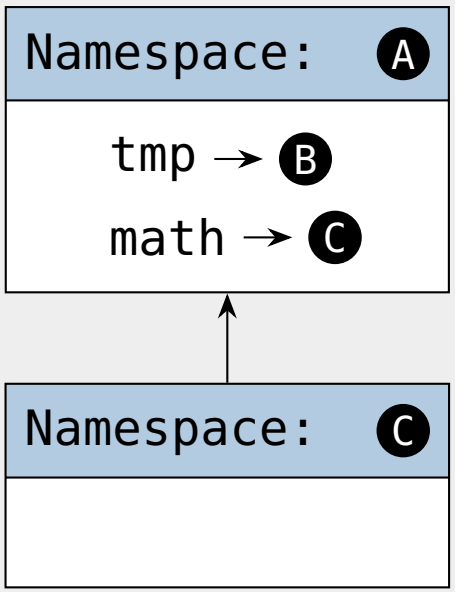
Symbol Table

Namespace: A
tmp → B

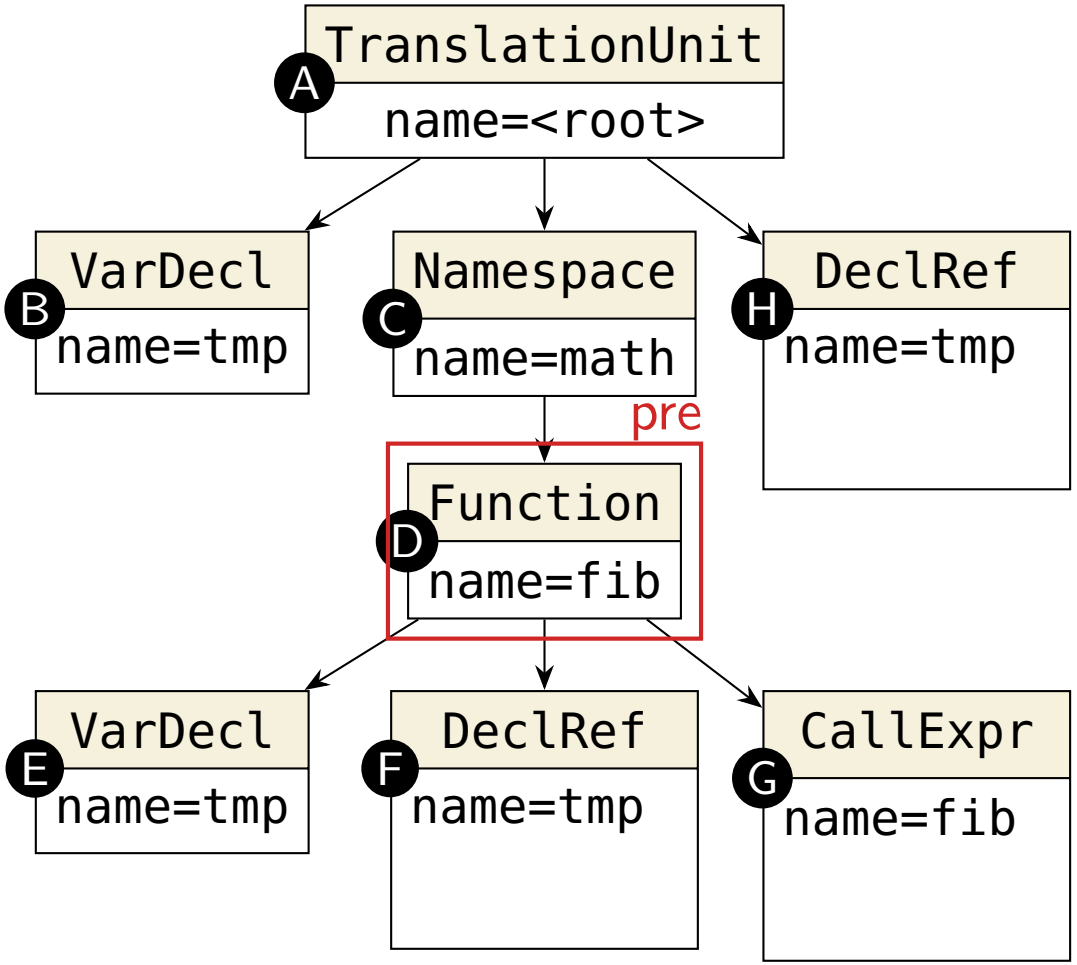
Symboltabelle als Stack von Namensräumen

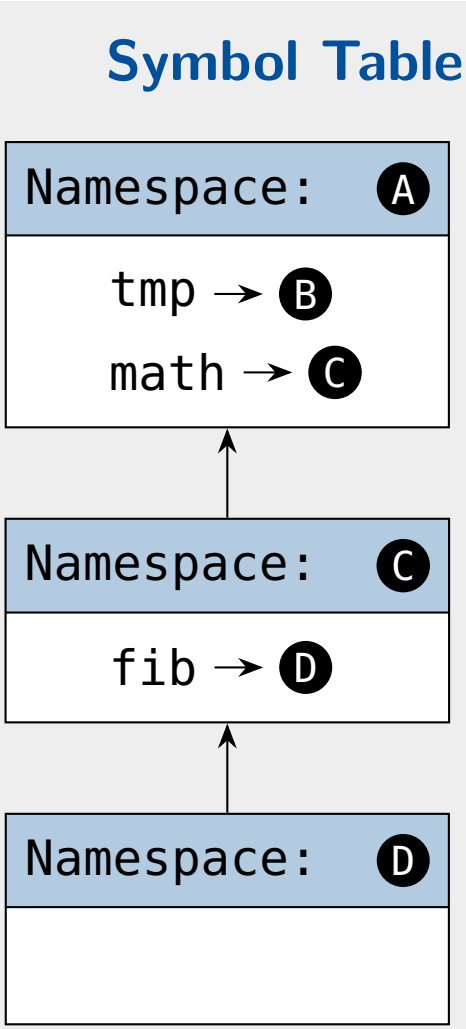


Symbol Table

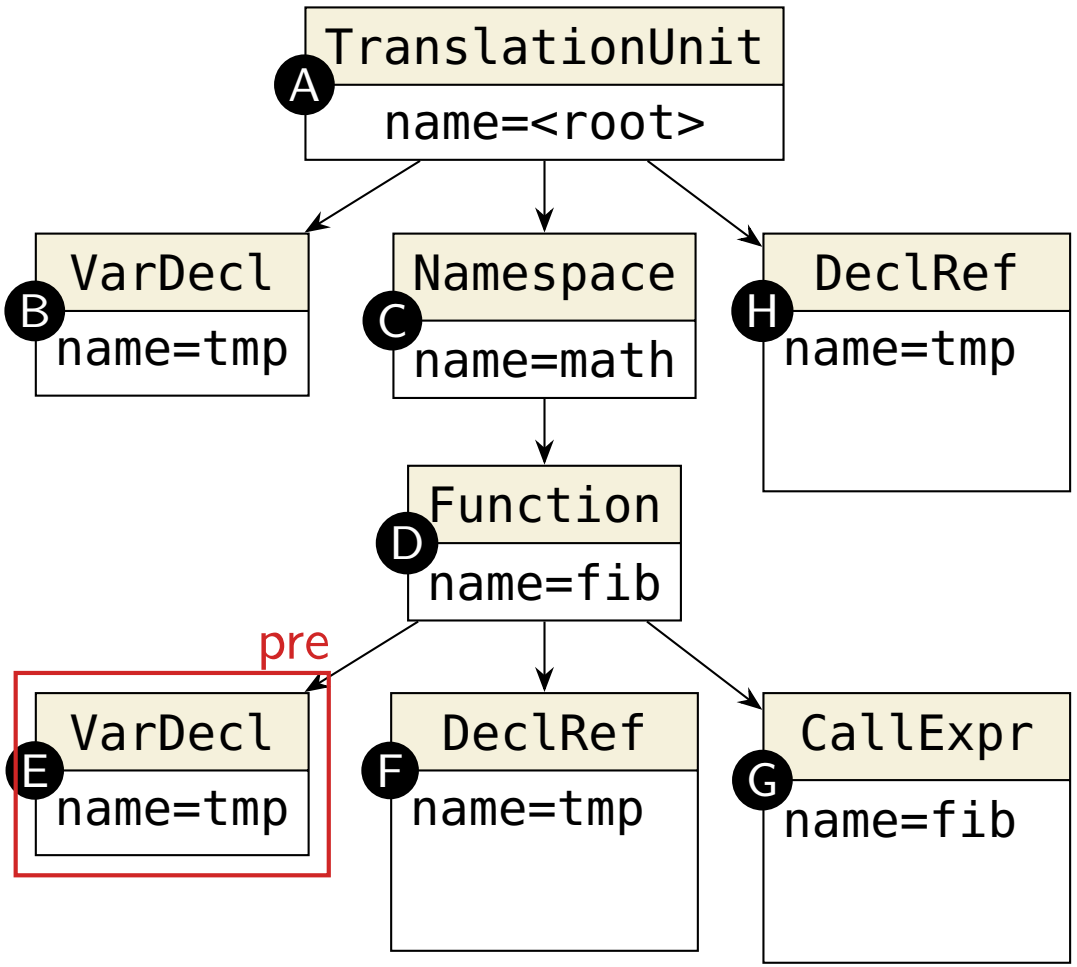


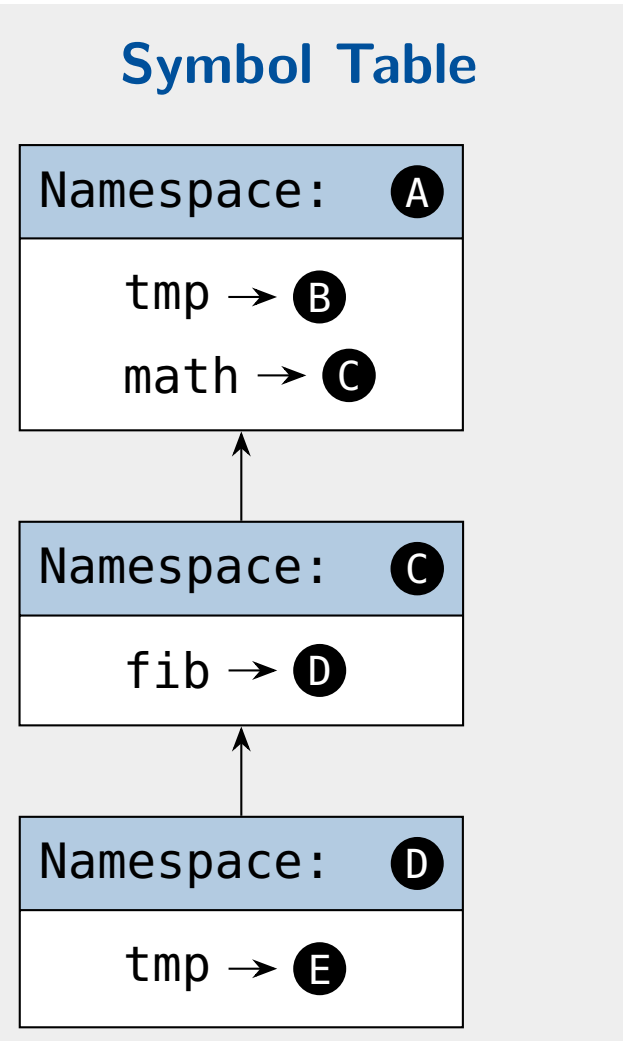
Symboltabelle als Stack von Namensräumen



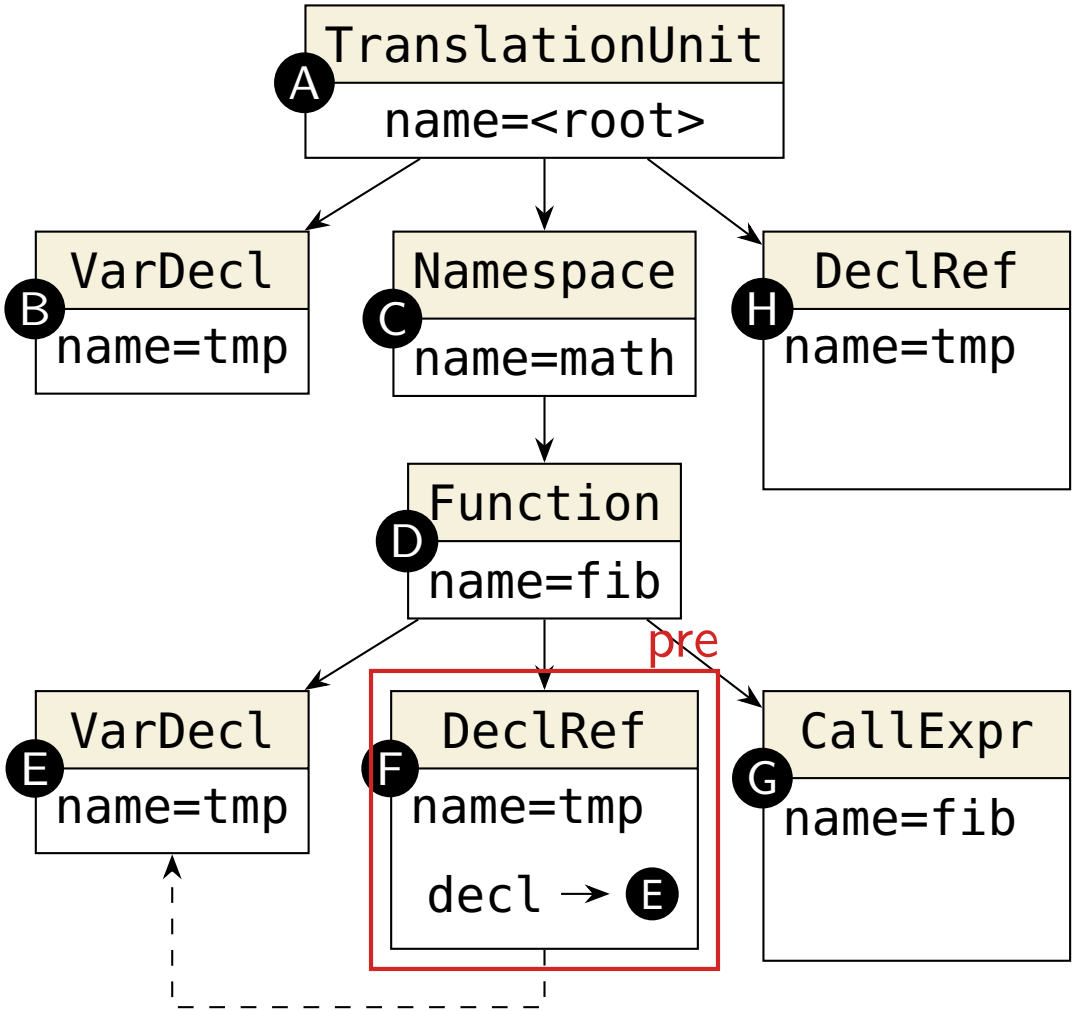


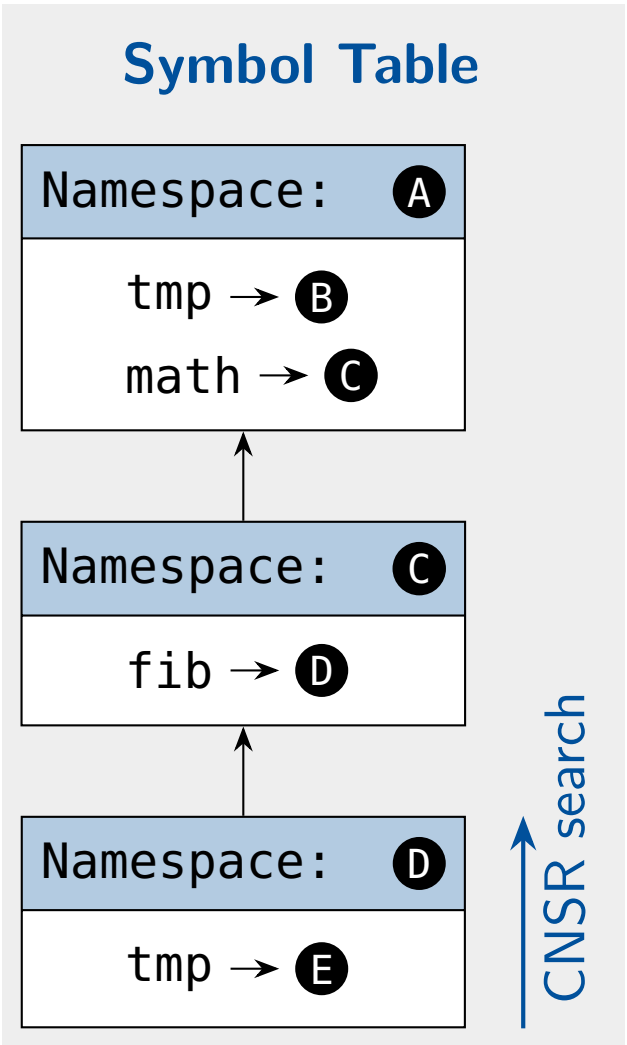
Symboltabelle als Stack von Namensräumen



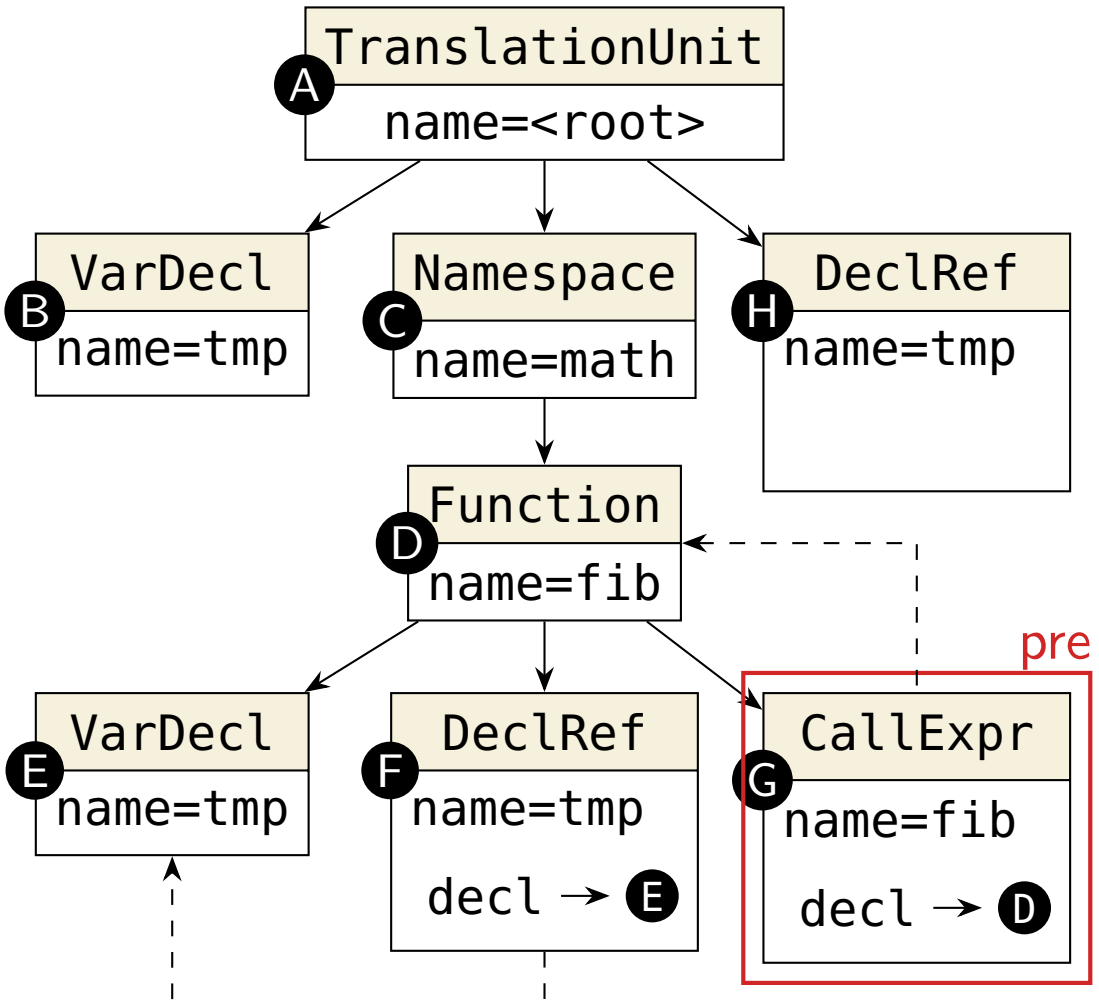


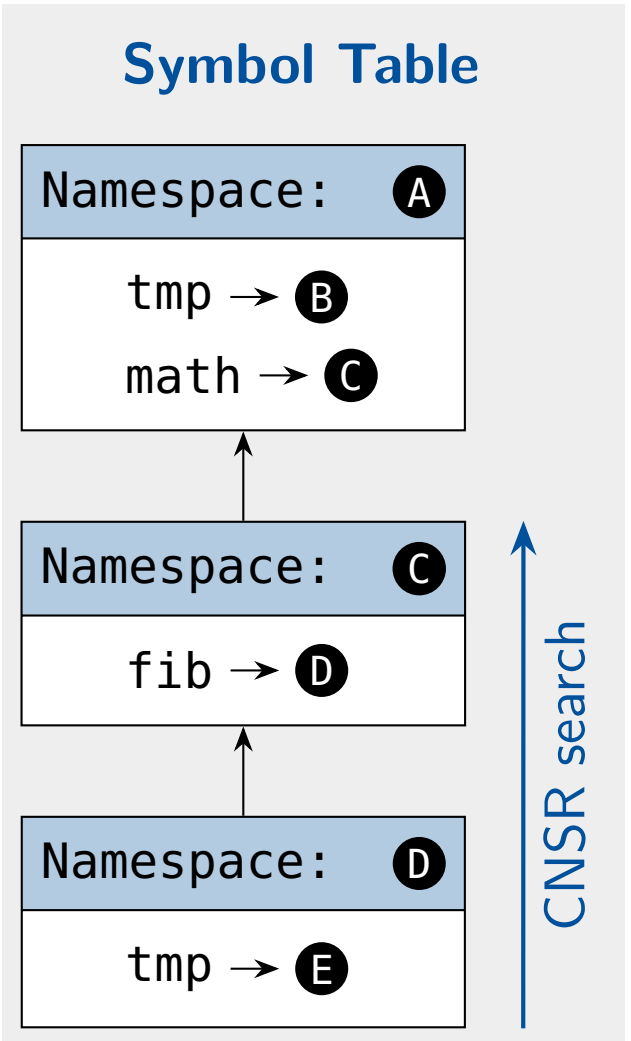
Symboltabelle als Stack von Namensräumen



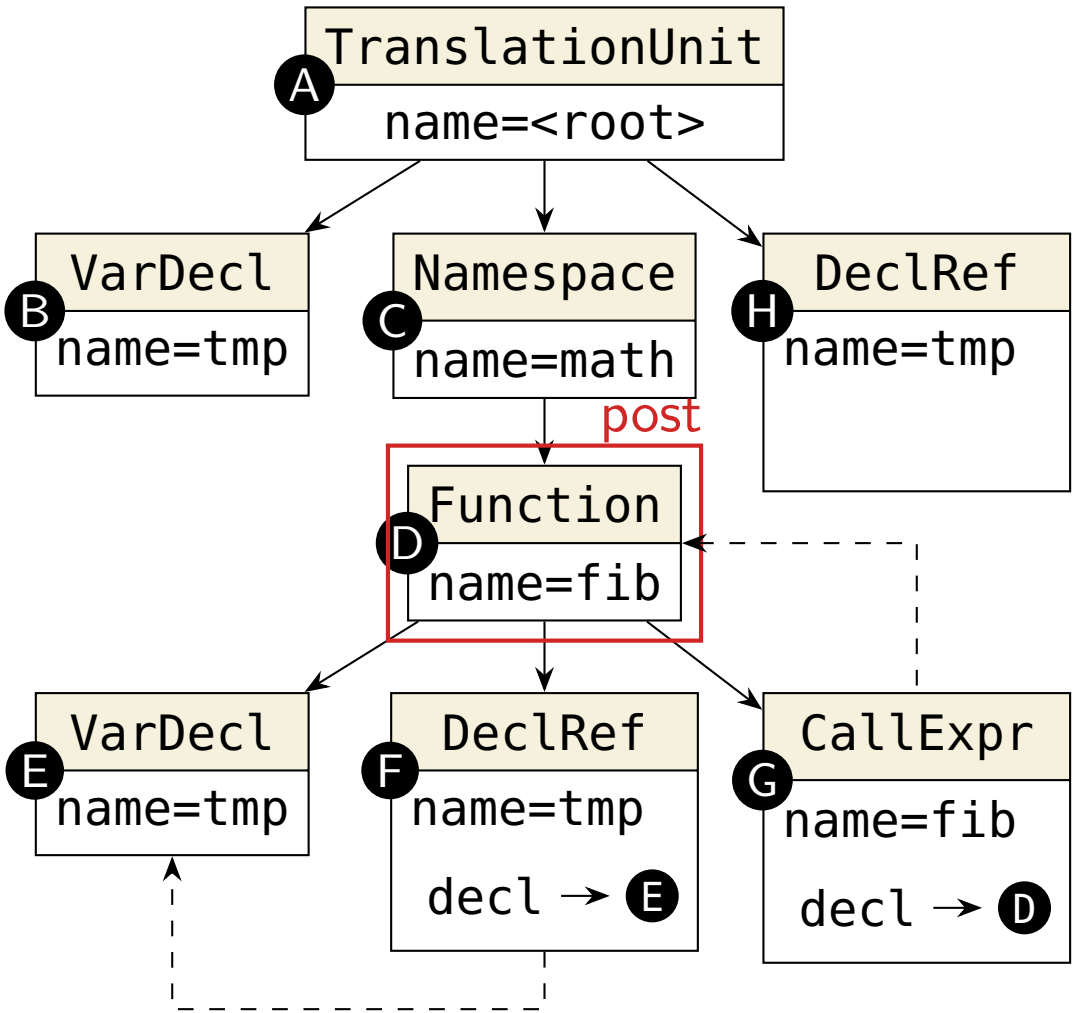


Symboltabelle als Stack von Namensräumen

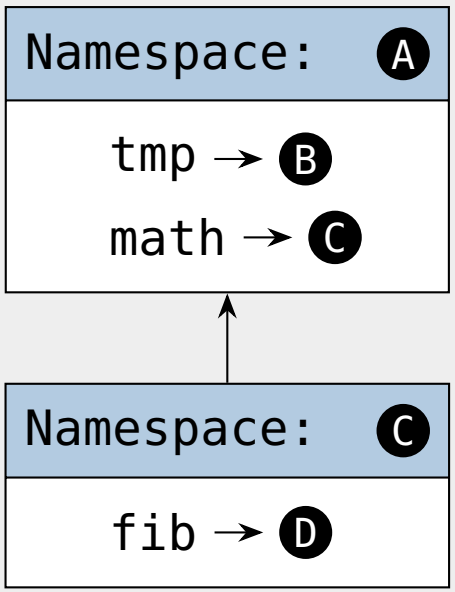




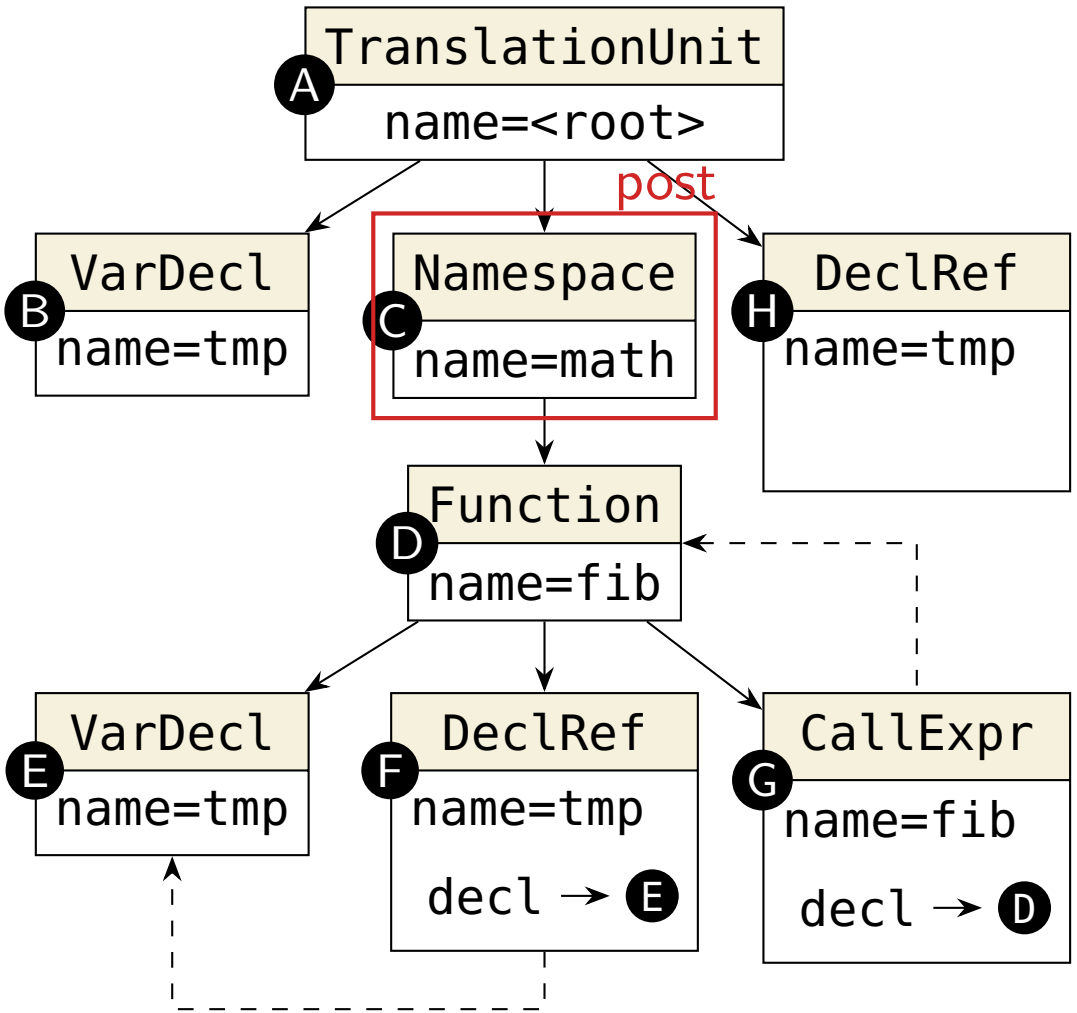
Symboltabelle als Stack von Namensräumen



Symbol Table



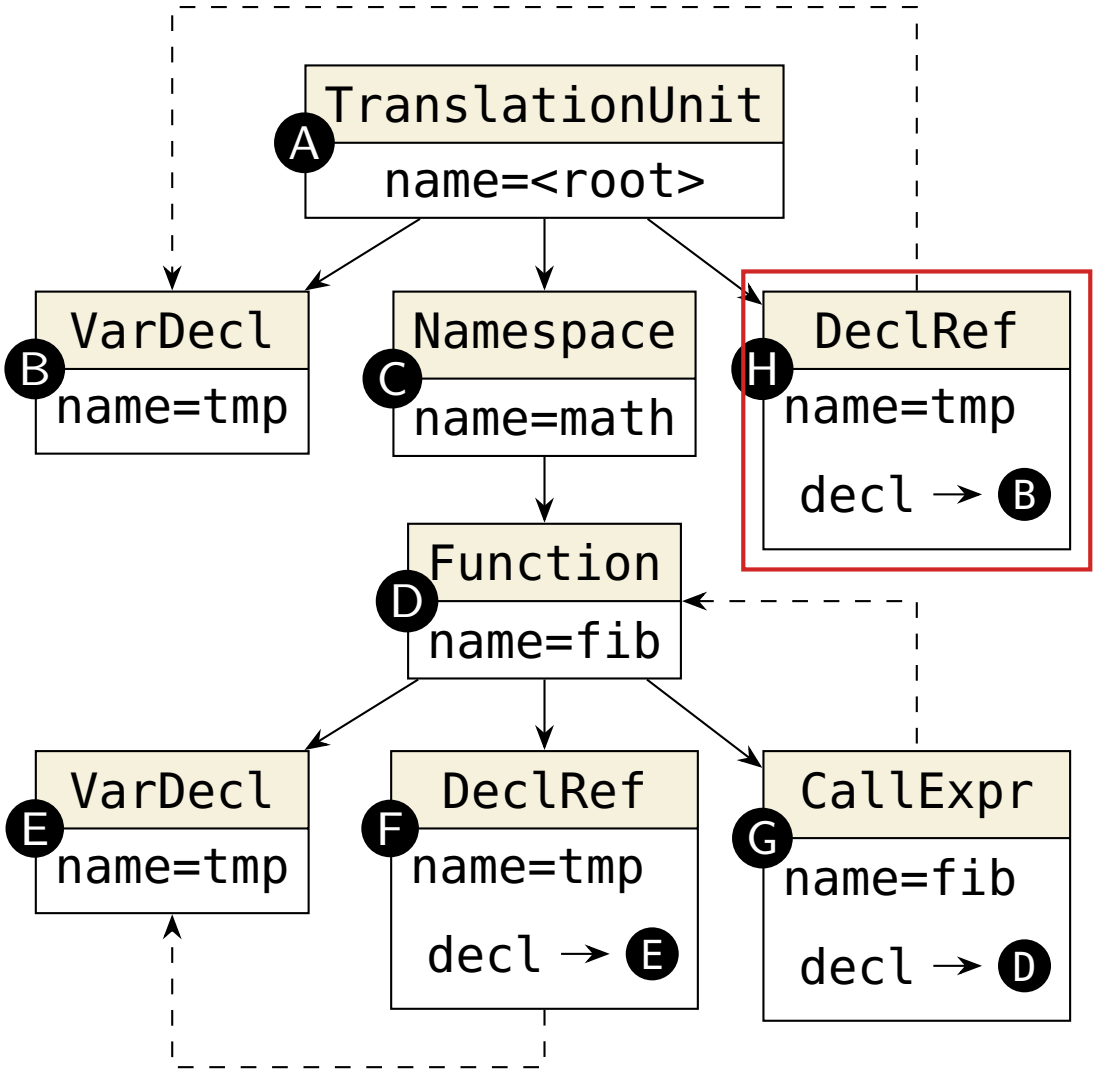
Symboltabelle als Stack von Namensräumen



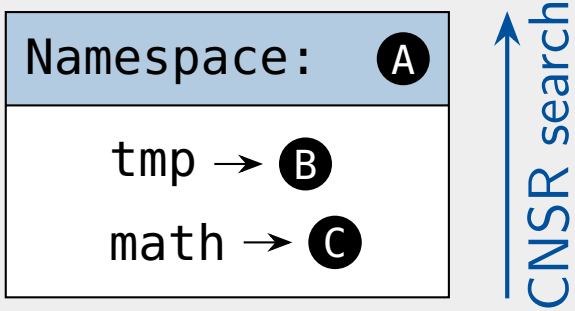
Symbol Table

Namespace: A
tmp → B
math → C

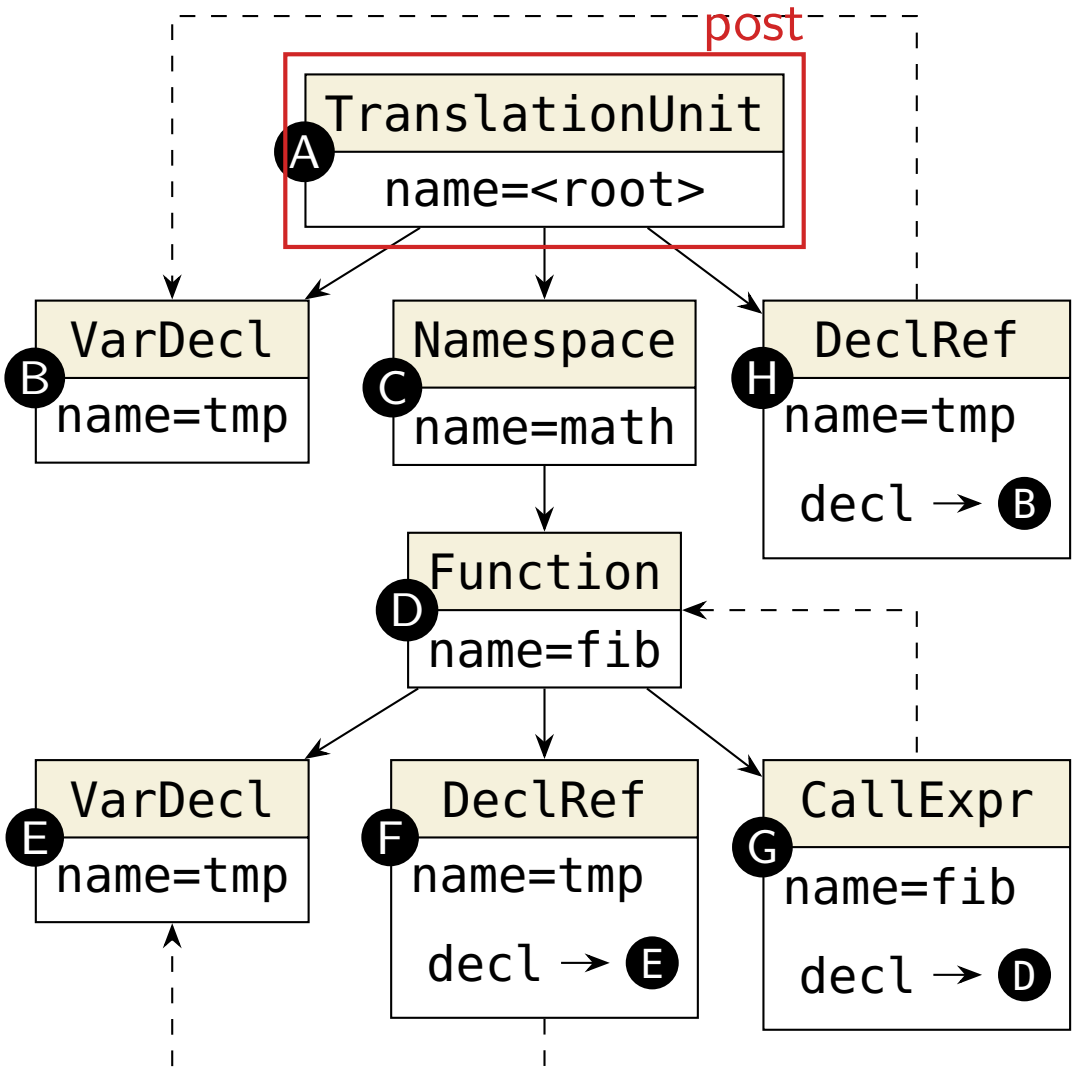
Symboltabelle als Stack von Namensräumen



Symbol Table



Symboltabelle als Stack von Namensräumen



Symbol Table

Erweiterte Traversierung bei der Namensauflösung

- Beliebige Reihenfolge von Deklaration und Referenzen in einem NS
  - Java: Reihenfolge der Methodendefinition ist egal!
  - **Lösung:** Wir traversieren den AST zweimal.
    1. Einsammeln aller Deklarationen, Speicherung an den Knoten (z.B. `cClassDecl`)
    2. Auflösen der Referenzen beim zweiten Durchlauf

Erweiterte Traversierung bei der Namensauflösung

- Beliebige Reihenfolge von Deklaration und Referenzen in einem NS
  - Java: Reihenfolge der Methodendefinition ist egal!
  - **Lösung:** Wir traversieren den AST zweimal.
    1. Einsammeln aller Deklarationen, Speicherung an den Knoten (z.B. `classDecl`)
    2. Auflösen der Referenzen beim zweiten Durchlauf

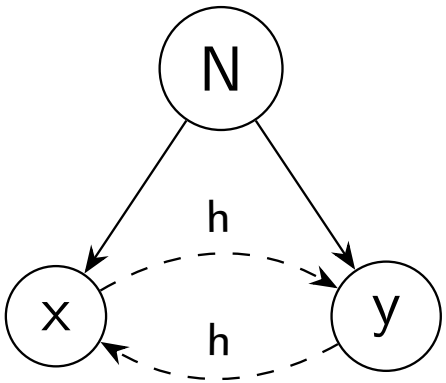
- Überladung von Funktionen anhand der statischen Signaturen
  - Typinformationen müssen bei der Namensauflösung bereitstehen
  - **Lösung**: Typberechnung und Namensauflösung im selben Visitor durchführen

```python
def post_CallExpr(self, call): # Post-Order: Bekannte Argumen
    # 1. Berechnung der statischen Signatur
    N = signature(call.func_name)
    for A in call.arguments:
      N.addParam(A.Type)

    # 2. Namensauflösung des komplexen Namens
    call.decl = self.ST.findName(N)

    # 3. Ergebnistyp der CallExpr
    call.Type = call.decl.ReturnType
```

Python

Zyklische Attribute und Unifikation

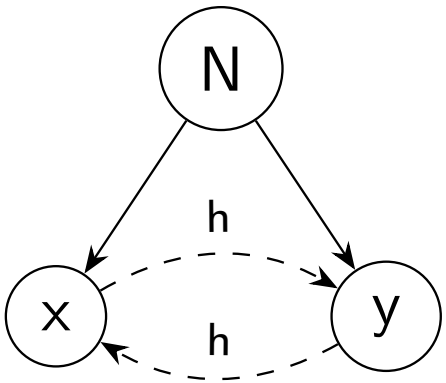
Zyklisch-abhängige Attribute



- Zyklische Attribute erfordern komplexen Datenfluss
  - Knoten liefern Teilwissen über das Attribut
  - Gleichungssystem für Attribute ist selbst-referenziell
  - Einfaches zyklisches Beispiel:

```
x.h = y.h - 1        Konsekutive Zahlen
y.h = 2 * x.h        Vielfachheit
```

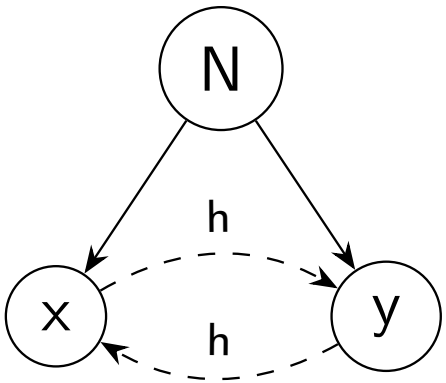
Zyklisch-abhängige Attribute



- Zyklische Attribute erfordern komplexen Datenfluss
  - Knoten liefern Teilwissen über das Attribut
  - Gleichungssystem für Attribute ist selbst-referenziell
  - Einfaches zyklisches Beispiel: `x.h = 1, y.h = 2`

```
x.h = y.h - 1        Konsekutive Zahlen
y.h = 2 * x.h        Vielfachheit
```

Zyklisch-abhängige Attribute

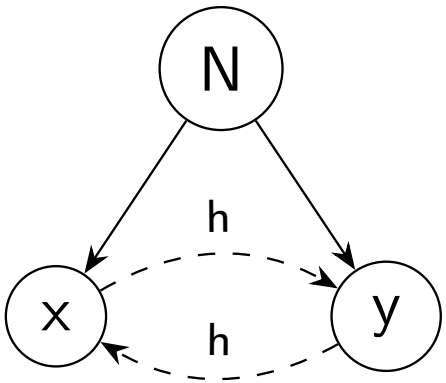


- Zyklische Attribute erfordern komplexen Datenfluss
  - Knoten liefern Teilwissen über das Attribut
  - Gleichungssystem für Attribute ist selbst-referenziell
  - Einfaches zyklisches Beispiel: `x.h = 1, y.h = 2`

    ```
    x.h = y.h - 1        Konsekutive Zahlen
    y.h = 2 * x.h        Vielfachheit
    ```

  - ... wurden lange für Programmiersprachen gemieden.

Zyklisch-abhängige Attribute



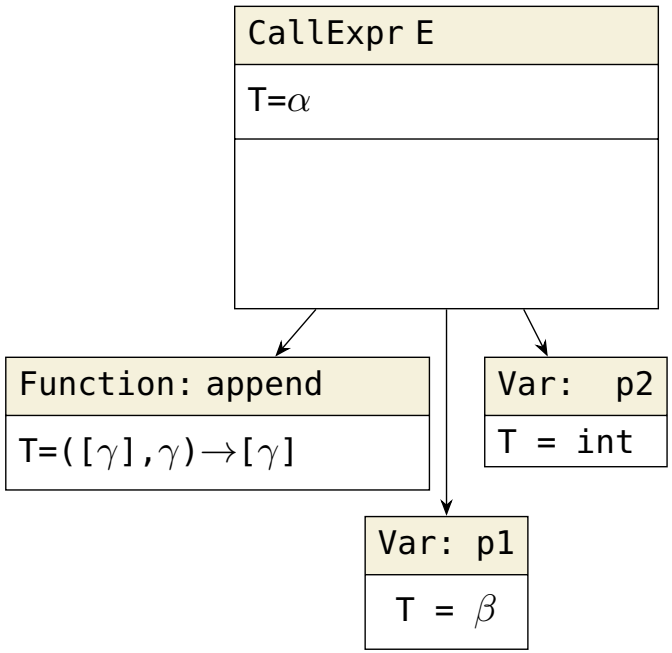
- Zyklische Attribute erfordern komplexen Datenfluss
  - Knoten liefern Teilwissen über das Attribut
  - Gleichungssystem für Attribute ist selbst-referenziell
  - Einfaches zyklisches Beispiel: `x.h = 1, y.h = 2`

    ```
    x.h = y.h - 1        Konsekutive Zahlen
    y.h = 2 * x.h        Vielfachheit
    ```

  - ... wurden lange für Programmiersprachen gemieden.

- Typinferenz erleichtert statische Typisierung, erzeugt aber Zyklen
  - Inferenz: Ableitung der statischen Typen aus der Verwendung der Variablen
  - `fun(x) { return x + 2 }` → Der Parameter x muss vom Typ `int` sein
  - Hindley-Milner-Typinferenz kombiniert zyklische Attribute mit **parametrischem Polymorphismus**; Grundlage der funktionalen Programmierung

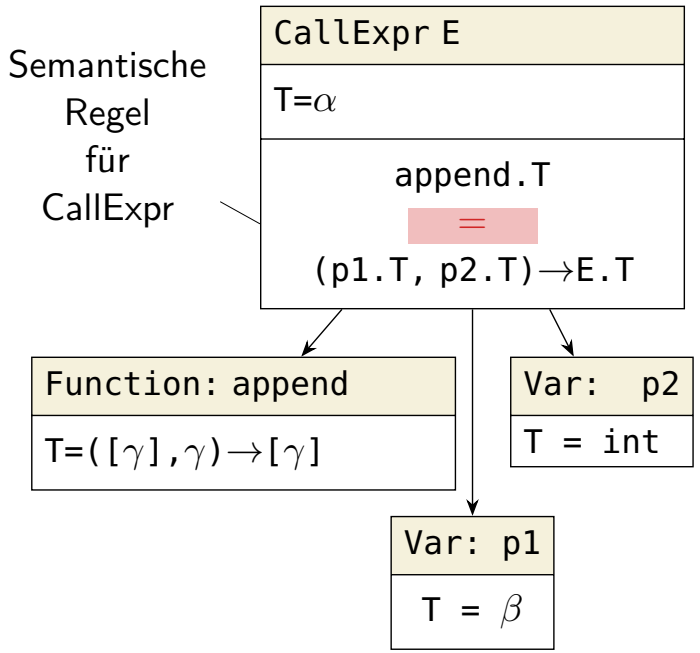
Typinferenz durch Ersetzung von Typvariablen



Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.

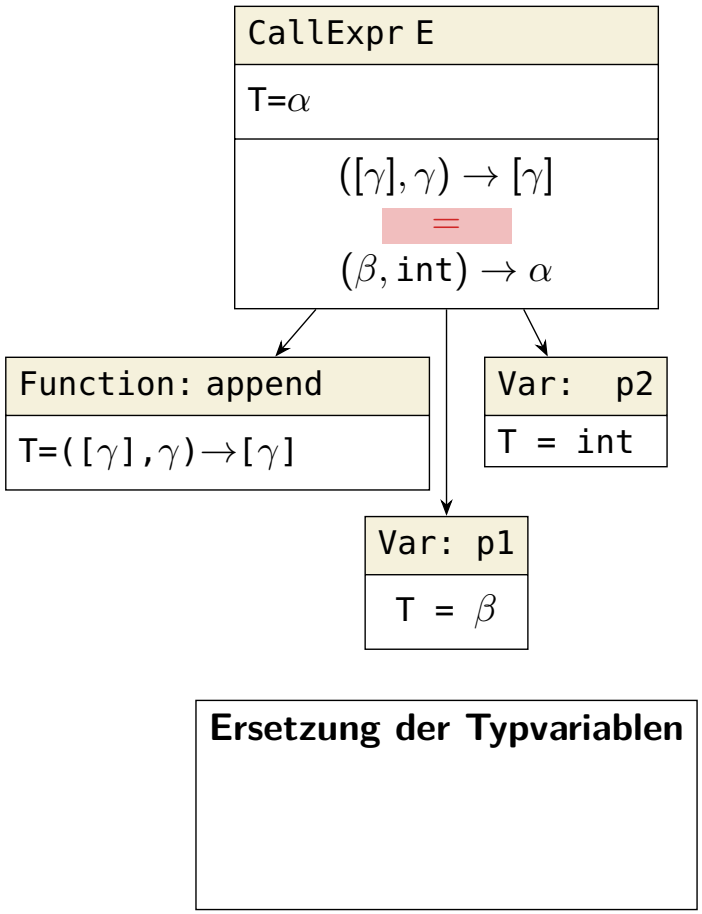
Typinferenz durch Ersetzung von Typvariablen



Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.
- Regeln als parametrische Bedingungen
  - CallExpr: Die Signatur der aufgerufenen Funktion muss zu den Argumenten passen.
  - Rückggabetyp (E.T) soll ermittelt werden

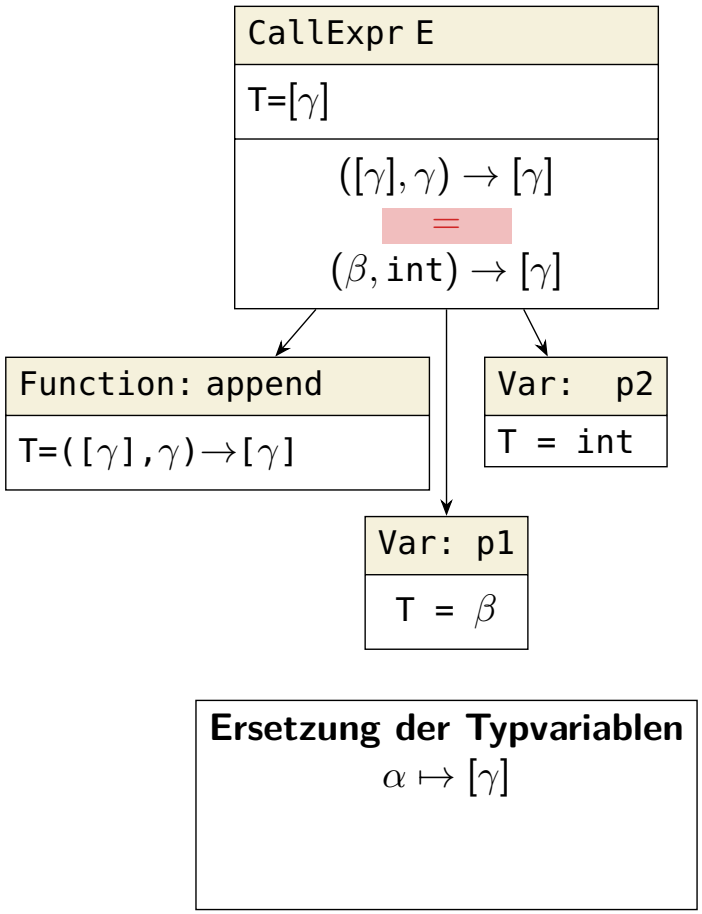
Typinferenz durch Ersetzung von Typvariablen



Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.
- Regeln als parametrische Bedingungen
  - CallExpr: Die Signatur der aufgerufenen Funktion muss zu den Argumenten passen.
  - Rückggabetyp (E.T) soll ermittelt werden
- Variablen-Elimination durch Ersetzungen
  - **Ziel:** Bedingungen müssen erfüllt sein
  - Ersetztung: Variable $\to$ Typausdruck
  - Finde Sequenz unifizierender Ersetzungen

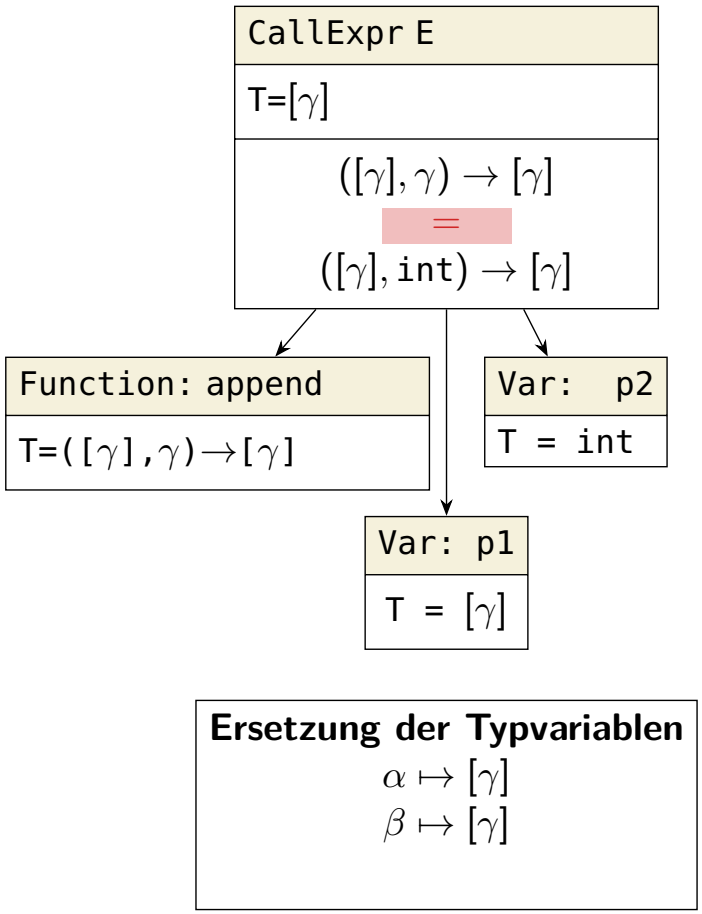
Typinferenz durch Ersetzung von Typvariablen



Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.
- Regeln als parametrische Bedingungen
  - CallExpr: Die Signatur der aufgerufenen Funktion muss zu den Argumenten passen.
  - Rückggabetyp (E.T) soll ermittelt werden
- Variablen-Elimination durch Ersetzungen
  - **Ziel:** Bedingungen müssen erfüllt sein
  - Ersetztung: Variable $\rightarrow$ Typausdruck
  - Finde Sequenz unifizierender Ersetzungen

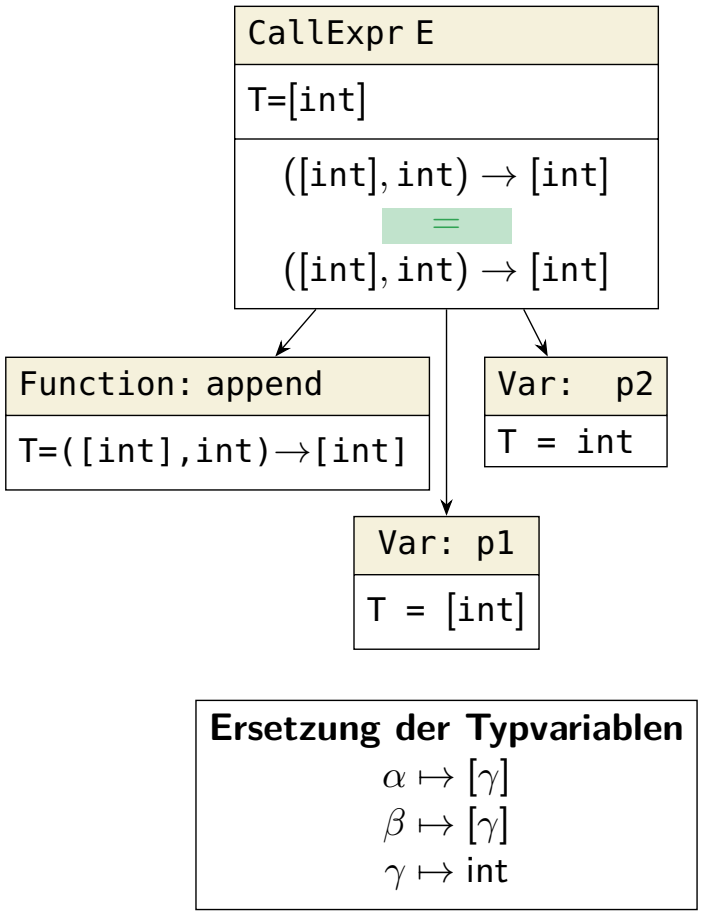
Typinferenz durch Ersetzung von Typvariablen



Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.
- Regeln als parametrische Bedingungen
  - CallExpr: Die Signatur der aufgerufenen Funktion muss zu den Argumenten passen.
  - Rückggabetyp (E.T) soll ermittelt werden
- Variablen-Elimination durch Ersetzungen
  - **Ziel:** Bedingungen müssen erfüllt sein
  - Ersetztung: Variable $\rightarrow$ Typausdruck
  - Finde Sequenz unifizierender Ersetzungen

Typinferenz durch Ersetzung von Typvariablen



Ersetzung der Typvariablen

$\alpha \mapsto [\gamma]$

$\beta \mapsto [\gamma]$

$\gamma \mapsto$ int

Was ist der Typ von `append(p1, p2)` ?

- Blätter: Einsetzen des vorhandenen Wissens
  - Die Variable p2 hat Typ `int`
  - `append()` ist generische Funktion und hängt ein Element an homogene Liste an
  - Alles Unbekannte wird durch eine freie Typvariable (α, β, γ) dargestellt.
- Regeln als parametrische Bedingungen
  - CallExpr: Die Signatur der aufgerufenen Funktion muss zu den Argumenten passen.
  - Rückggabetyp (E.T) soll ermittelt werden
- Variablen-Elimination durch Ersetzungen
  - **Ziel:** Bedingungen müssen erfüllt sein
  - Ersetztung: Variable → Typausdruck
  - Finde Sequenz **unifizierender Ersetzungen**

Unifikation: Berechnung von passenden Ersetzungen

Unifikation

Finden einer Sequenz von Variablen-Ersetzungen, die zwei Terme angleicht.

$$T_1 = \text{func}(\text{list}(\gamma), \gamma, \quad \text{list}(\gamma))$$
$$T_2 = \text{func}(\beta, \quad \text{int}, \ \alpha)$$

Typausdrücke mit freien Variablen

- Typ(konstruktoren): int, func, list
- Variablen: α, β, γ
- Variablen können in beiden Termen, auch mehrfach, vorkommen

Unifikation: Berechnung von passenden Ersetzungen

Unifikation

Finden einer Sequenz von Variablen-Ersetzungen, die zwei Terme angleicht.

$$T_1 = \text{func}(\text{list}(\gamma), \gamma, \quad \text{list}(\gamma))$$
$$T_2 = \text{func}(\beta, \quad \text{int}, \quad \alpha)$$

$$U = \{\underbrace{\alpha \mapsto \beta}_{S_1}, \underbrace{\beta \mapsto \mathit{list}(\gamma)}_{S_2}, \underbrace{\gamma \mapsto \text{int}}_{S_3}\}$$

Typausdrücke mit freien Variablen

- Typ(konstruktoren): int, func, list
- Variablen: α, β, γ
- Variablen können in beiden Termen, auch mehrfach, vorkommen

Unifizierende Ersetzung

- Ersetzungssequenz: $S_1 \circ S_2 \circ S_3$
- Reihenfolge ist relevant!
- **Allgemeinster Unifikator** nimmt nur notwendige Ersetzungen vor

Unifikation: Berechnung von passenden Ersetzungen

Unifikation

Finden einer Sequenz von Variablen-Ersetzungen, die zwei Terme angleicht.

$$T_1 = \text{func}(\text{list}(\gamma), \gamma, \quad \text{list}(\gamma))$$
$$T_2 = \text{func}(\beta, \quad \text{int}, \quad \alpha)$$

$$U = \{\underbrace{\alpha \mapsto \beta}_{S_1}, \underbrace{\beta \mapsto \mathit{list}(\gamma)}_{S_2}, \underbrace{\gamma \mapsto \text{int}}_{S_3}\}$$

Typausdrücke mit freien Variablen

- Typ(konstruktoren): int, func, list
- Variablen: α, β, γ
- Variablen können in beiden Termen, auch mehrfach, vorkommen

Unifizierende Ersetzung

- Ersetzungssequenz: $S_1 \circ S_2 \circ S_3$
- Reihenfolge ist relevant!
- **Allgemeinster Unifikator** nimmt nur notwendige Ersetzungen vor

- Mehrere Unifikations-Algorithmen $\texttt{unify}(T_1, T_2) \rightarrow U$
  - Erster Algorithmus von Robinson (1965) hat exponentielle Laufzeit
  - Lineare Laufzeit: Paterson, Wegman (1978); Martelli, Montanari (1982)

Zusammenfassung: Semantische Analyse

Technische Universität Braunschweig

- Semantische Analyse prüft die restlichen kontextsensitiven Sprachregeln
  - Namensauflösung: Gibt es zu jeder Referenz eine Deklaration?
  - Typkonsistenz: Werden die statischen Typen korrekt verwendet?

$\Rightarrow$ Nach der semantischen Analyse haben wir ein korrektes Programm vor uns!

Zusammenfassung: Semantische Analyse

Technische Universität Braunschweig

- Semantische Analyse prüft die restlichen kontextsensitiven Sprachregeln
  - Namensauflösung: Gibt es zu jeder Referenz eine Deklaration?
  - Typkonsistenz: Werden die statischen Typen korrekt verwendet?

  $\Rightarrow$ Nach der semantischen Analyse haben wir ein korrektes Programm vor uns!

- Berechnung der semantischen Attribute für jeden Knoten (z.B. Typ)
  - Attribute sind Abhängig von anderen (auch entfernten) Knotenattributen
    $\Rightarrow$ Gleichungssystem zwischen allen Knotenattributen
  - S-Attribut: Nur Abhängigkeiten zu den eigenen Kindern („nach unten“)
  - L-Attribut: Abhängigkeiten von Geschwister und Vorgängern („nach links“)
  - Zyklische Attribute: Wechselseitig abhängige Attributgleichungen

Zusammenfassung: Semantische Analyse

- Semantische Analyse prüft die restlichen kontextsensitiven Sprachregeln
  - Namensauflösung: Gibt es zu jeder Referenz eine Deklaration?
  - Typkonsistenz: Werden die statischen Typen korrekt verwendet?

  $\Rightarrow$ Nach der semantischen Analyse haben wir ein korrektes Programm vor uns!

- Berechnung der semantischen Attribute für jeden Knoten (z.B. Typ)
  - Attribute sind Abhängig von anderen (auch entfernten) Knotenattributen
    $\Rightarrow$ Gleichungssystem zwischen allen Knotenattributen
  - S-Attribut: Nur Abhängigkeiten zu den eigenen Kindern („nach unten“)
  - L-Attribut: Abhängigkeiten von Geschwister und Vorgängern („nach links“)
  - Zyklische Attribute: Wechselseitig abhängige Attributgleichungen

- Übersetzertechniken zur Attributberechnung und Regelprüfung
  - Baumtraversierung: Visitor bündelt knotentypabhängige Operationen
  - Symboltabelle: Speichert die aktuell deklarierten Namen beim Traversieren
  - Unifikation: Angleichung von parametrischen Typausdrücken durch Ersetzung