

## Beispiel:

```
public class Foo {  
    protected int foo() { return 1; }  
}  
class Fee extends Foo {  
    protected int foo() { return 2; }  
    public int test(boolean b) {  
        Foo x = (b) ? new Foo() : new Fee();  
        return x.foo();  
    }  
}
```

**Aufgabe:** Finde zu jeder Benutzung eines Bezeichners die zugehörige Definition

**1. Schritt:** Ersetze Bezeichner durch **eindeutige** Nummern !

**Input:** Folge von Strings

**Output:** (1) Folge von Nummern  
(2) Tabelle, die zu Nummern die Strings auflistet

**Beispiel:**

das	schwein	ist	dem	schwein	was	...
-----	---------	-----	-----	---------	-----	-----

...	das	schwein	dem	menschen	ist	wurst
-----	-----	---------	-----	----------	-----	-------

... liefert:

0	1	2	3	1	4	0	1	3	5	2	6
---	---	---	---	---	---	---	---	---	---	---	---

0	das
1	schwein
2	ist
3	dem
4	was
5	menschen
6	wurst

## Implementierung 1:

Wir benutzen eine **partielle Abbildung**:  $S : \text{String} \rightarrow \text{int}$  verwaltet :-)

Wir verwalten einen Zähler  $\text{int count} = 0$ ; für die Anzahl der bereits gefundenen Wörter :-)

Damit definieren wir eine Funktion:  $\text{int getIndex}(\text{String } w)$  :

```
int getIndex(String  $w$ ) {  
    if ( $S(w) \equiv \text{undefined}$ ) {  
         $S = S \oplus \{w \mapsto \text{count}\}$ ;  
        return  $\text{count}++$ ;  
    } else return  $S(w)$ ;  
}
```

## Implementierung 2: Partielle Abbildungen

### Ideen:

- Liste von Paaren  $(w, i) \in \text{String} \times \text{int}$  :
  - Einfügen:  $\mathcal{O}(1)$
  - Finden:  $\mathcal{O}(n) \implies$  zu teuer :-)
- balancierte Bäume :
  - Einfügen:  $\mathcal{O}(\log(n))$
  - Finden:  $\mathcal{O}(\log(n)) \implies$  zu teuer :-)
- Hash Tables :
  - Einfügen:  $\mathcal{O}(1)$
  - Finden:  $\mathcal{O}(1) \dots$  zumindest im Mittel :-)

## ... im Beispiel:

- Wir legen ein Feld  $M$  von hinreichender Größe  $m$  an :-)
- Wir wählen eine Hash-Funktion  $H: \text{String} \rightarrow [0, m - 1]$  mit den Eigenschaften:
  - $H(w)$  ist leicht zu berechnen :-)
  - $H$  streut die vorkommenden Wörter gleichmäßig über  $[0, m - 1]$  :-)

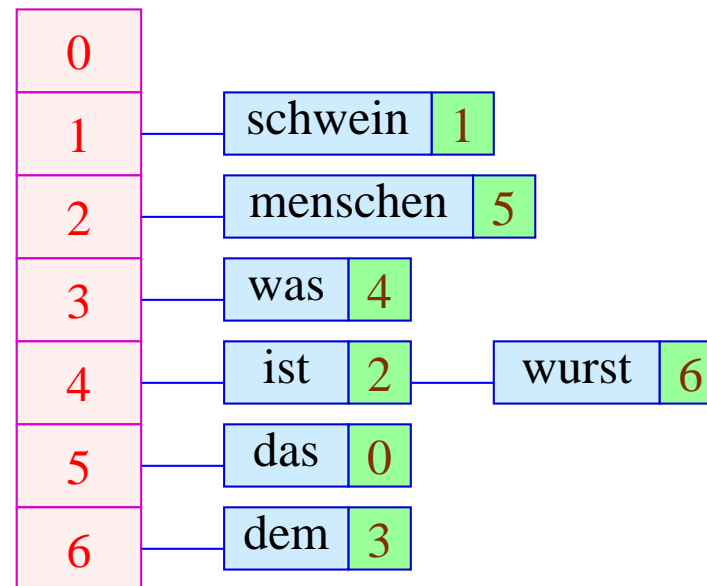
Mögliche Wahlen:

$$\begin{aligned} H_0(x_0 \dots x_{r-1}) &= (x_0 + x_{r-1}) \% m \\ H_1(x_0 \dots x_{r-1}) &= (\sum_{i=0}^{r-1} x_i \cdot p^i) \% m \\ &= (x_0 + p \cdot (x_1 + p \cdot (\dots + p \cdot x_{r-1} \dots))) \% m \end{aligned}$$

für eine Primzahl  $p$  (z.B. 31 :-)

- Das Argument-Wert-Paar  $(w, i)$  legen wir dann in  $M[H(w)]$  ab :-)

Mit  $m = 7$  und  $H_0$  erhalten wir:



Um den Wert des Worts  $w$  zu finden, müssen wir  $w$  mit allen Worten  $x$  vergleichen, für die  $H(w) = H(x)$  :-)

## 2. Schritt: Symboltabellen

- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
  - jede Definition **vor** ihren Benutzungen besucht :-)
  - die jeweils aktuell sichtbare Definition zuletzt besucht :-)
- Für jeden Bezeichner verwaltet man einen **Keller** der gültigen Definitionen.
- Trifft man bei der Durchmusterung auf eine Definition eines Bezeichners, schiebt man sie auf den Keller.
- Verlässt man den Gültigkeitsbereich, muss man sie wieder vom Keller werfen :-)
- Trifft man bei der Durchmusterung auf eine Benutzung, schlägt man die letzte Definition auf dem Keller nach ...
- Findet man keine Definition, haben wir einen Fehler gefunden :-)

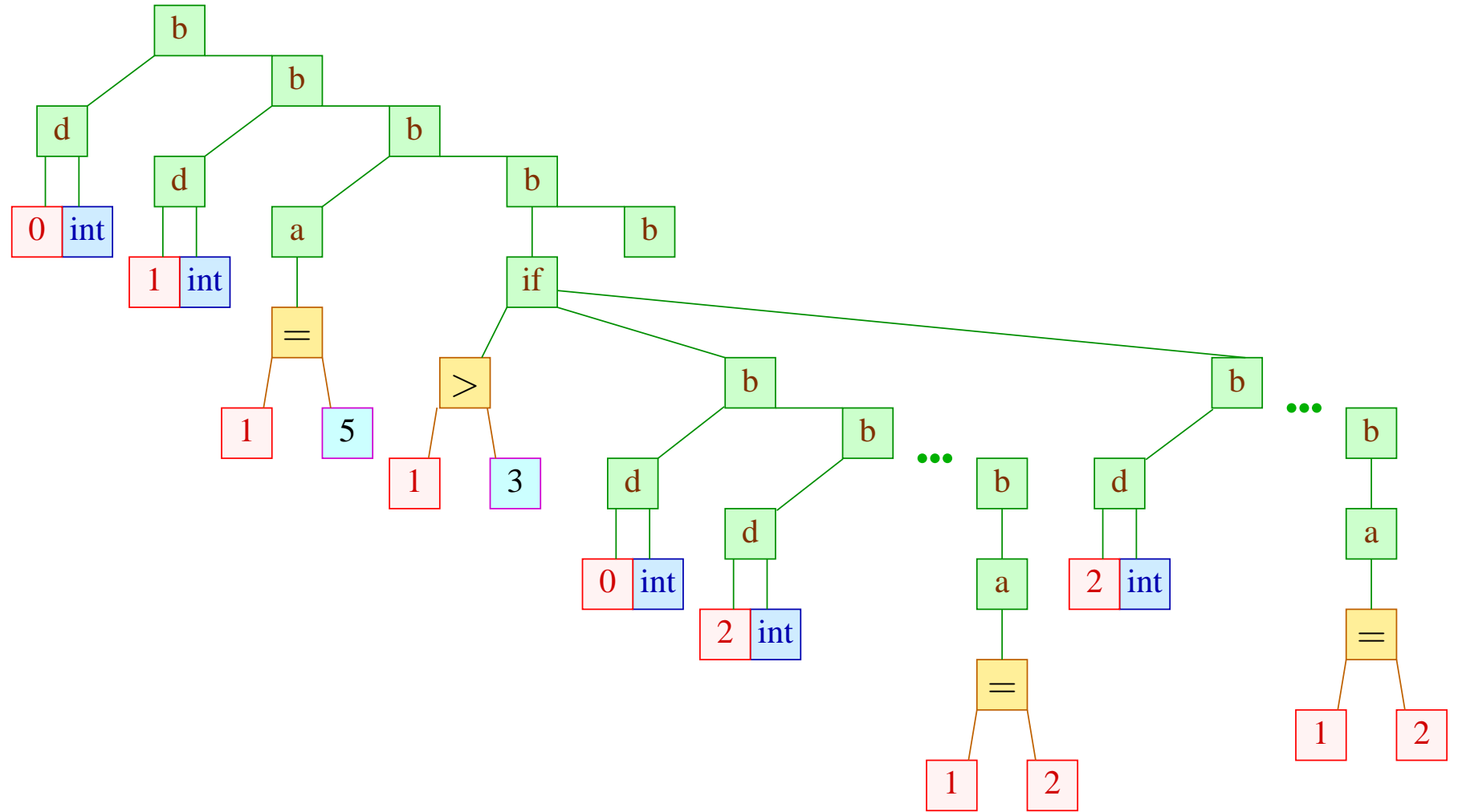


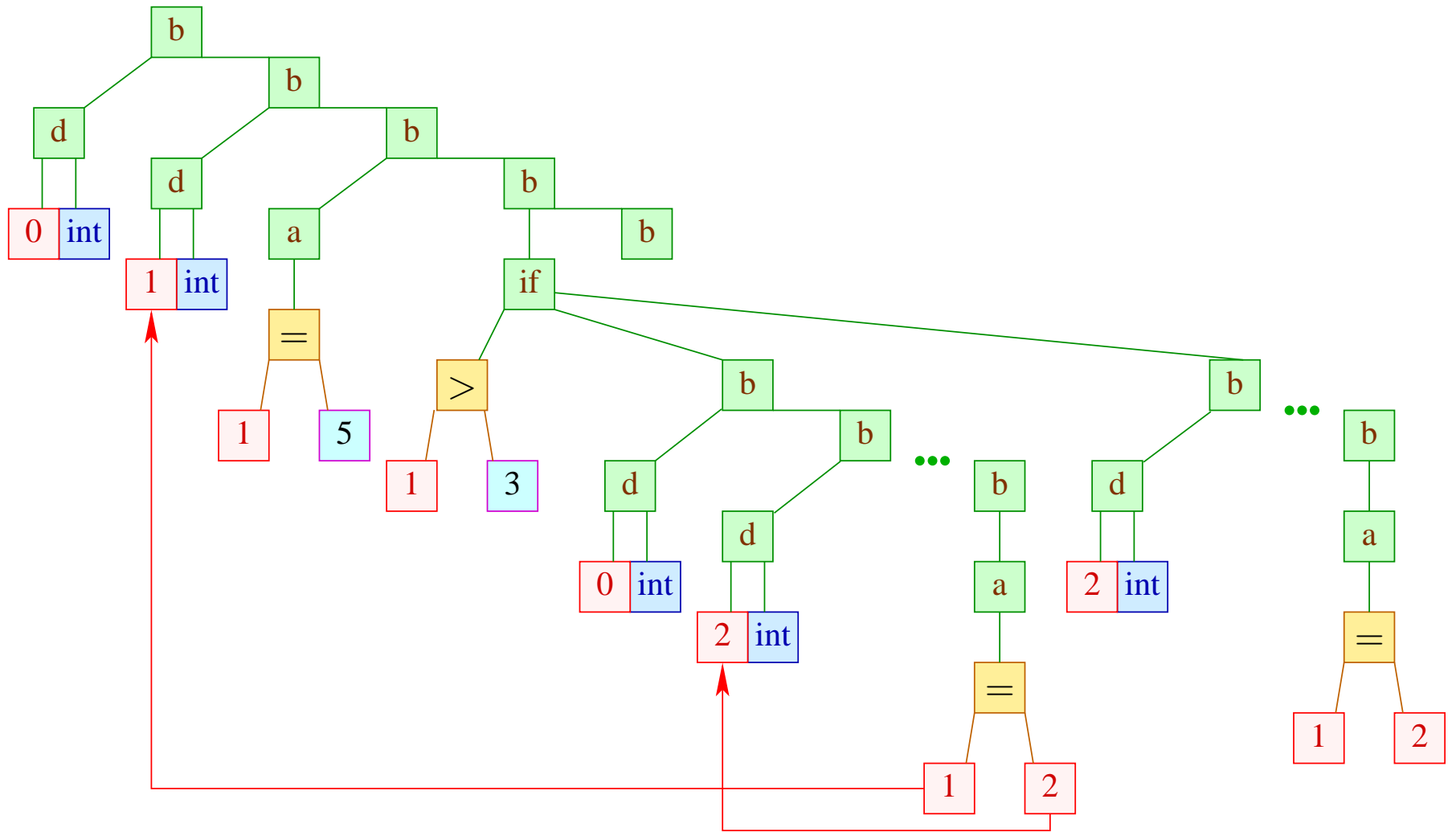
## Beispiel:

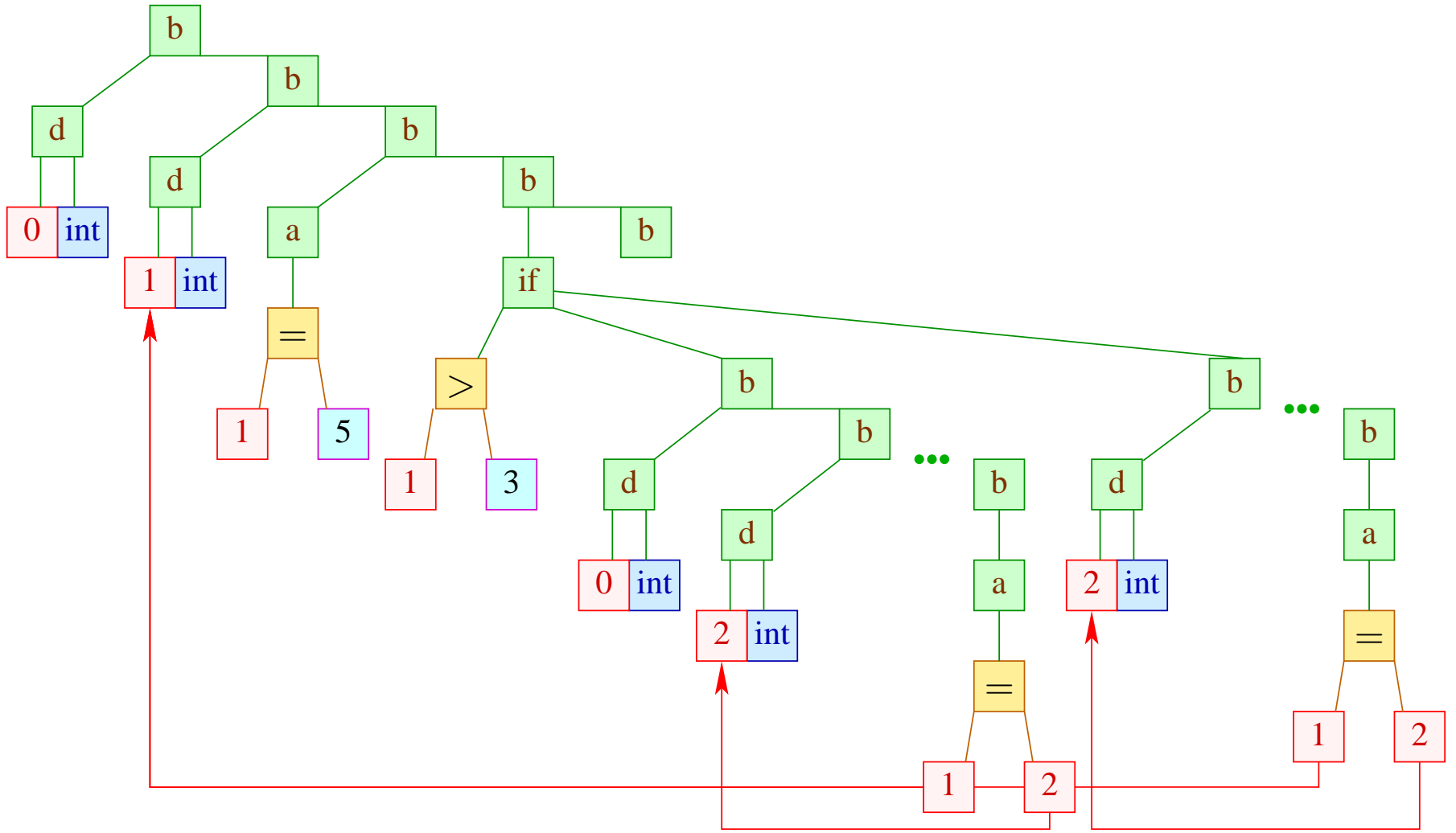
```
{ int a, b;
  a = 5;
  if (a > 3) {
    int a, c;
    a = 3;
    c = a + 1;
    b = c;
  }
} else {
  int c;
  c = a + 1;
  b = c;
}
```

0	<i>a</i>
1	<i>b</i>
2	<i>c</i>

Der zugehörige [Syntaxbaum ...](#)







## Diskussion:

- Der Durchlauf ist hier einfach **links-rechts** DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

## Diskussion:

- Der Durchlauf ist hier einfach **links-rechts** DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

## Achtung:

- Manche Programmiersprachen verbieten eine Mehrfach-Deklaration des selben Namens innerhalb eines Blocks ;-)
- Dann muss man für jede Deklaration einen Pointer auf den Block verwalten, zu dem sie gehört.
- Gibt es eine weitere Deklaration des gleichen Namens mit dem selben Pointer, muss ein Fehler gemeldet werden :-))

## Erweiterung:

- Hat man mehrere wechselseitig **rekursive Funktionsdefinitionen** in einem Block, müssen deren Namen vor Durchmustern der Rümpfe in die Tabelle eingetragen werden ...

```
fun  odd 0  = false
      |  odd 1  = true
      |  odd x  = even (x - 1)
and  even 0  = true
      |  even 1  = false
      |  even x  = odd (x - 1)
```

- Hat man eine objektorientierte Sprache mit Vererbung zwischen Klassen, sollte die übergeordnete Klasse vor der Unterklasse besucht werden :-)
- Bei Überladung muss simultan eine Typüberprüfung vorgenommen werden ...

## 3.2 Typ-Überprüfung

In modernen (imperativen / objektorientierten / funktionalen) Programmiersprachen besitzen Variablen und Funktionen einen **Typ**, z.B. **int**, **struct { int x; int y; }**.

Typen sind nützlich für:

- die **Speicherverwaltung**;
- die Vermeidung von **Laufzeit-Fehlern** :-)

In imperativen / objektorientierten Programmiersprachen muss der Typ bei der Deklaration spezifiziert und vom Compiler die typ-korrekte Verwendung überprüft werden :-)



Typen werden durch Typ-**Ausdrücke** beschrieben.

Die Menge  $T$  der Typausdrücke enthält:

- (1) **Basis-Typen:** **int, boolean, float, void** ...
- (2) **Typkonstruktoren**, angewendet auf Typen, z.B.:

- Verbunde: **struct** {  $t_1 a_1; \dots t_k a_k; \}$
- Zeiger:  $t *$
- Felder:  $t []$

**Achtung:**

In **C** muss zusätzlich eine Größe spezifiziert werden; die Variable muss dann zwischen  $t$  und  $[n]$  stehen **:-**(

- Funktionen:  $t (t_1, \dots, t_k)$

**Achtung:**

In **C** muss die Variable zwischen  $t$  und  $(t_1, \dots, t_k)$  stehen.

In **SML** dagegen würde man diesen Typ anders herum schreiben:

$t_1 * \dots * t_k \rightarrow t$  **:-**)

Wir benutzen:  $(t_1, \dots, t_k)$  als Tupel-Typen.

### (3) Typ-Namen.

Typ-Namen sind nützlich:

- als Abkürzung :-)

In C kann man diese mithilfe von **typedef** einführen:

```
typedef t x;
```

- zur Konstruktion rekursiver Typen ...

### Beispiel:

```
struct list0 {  
    int info;  
    struct list1 * next;  
};
```

```
struct list1 {  
    int info;  
    struct list0 * next;  
};
```

## Aufgabe:

**Gegeben:** eine Menge von Typ-Deklarationen  $\Gamma = \{t_1 x_1; \dots t_m x_m\}$

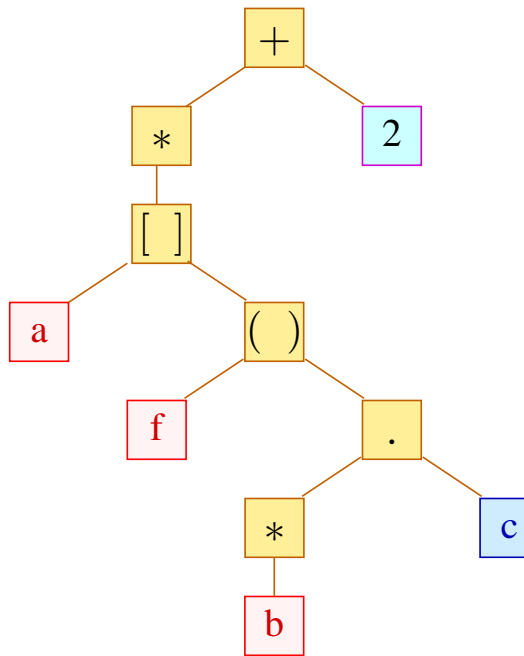
**Überprüfe:** Kann ein Ausdruck  $e$  mit dem Typ  $t$  versehen werden?

## Beispiel:

```
struct list {int info; struct list * next; };  
int f(struct list * l) {return 1; };  
struct {struct list * c; } * b;  
int * a[11];
```

Betrachte den Ausdruck:

$*a[f(b \rightarrow c)] + 2;$



## Idee:

- Traversiere den Syntaxbaum **bottom-up**.
- Für Bezeichner sagt uns  $\Gamma$  den richtigen Typ :-)
- Konstanten wie 2 oder 0.5 sehen wir den Typ direkt an ;-)
- Die Typen für die inneren Knoten erschießen wir mithilfe von **Typ-Regeln**.

Formal betrachten wir Aussagen der Form:

$$\Gamma \vdash e : t$$

// (In der Typ-Umgebung  $\Gamma$  hat  $e$  den Typ  $t$ )

Axiome:

Const:  $\Gamma \vdash c : t_c$  ( $t_c$  Typ der Konstante  $c$ )

Var:  $\Gamma \vdash x : \Gamma(x)$  ( $x$  Variable)

Regeln:

$$\text{Ref: } \frac{\Gamma \vdash e : t}{\Gamma \vdash \&e : t^*}$$

$$\text{Deref: } \frac{\Gamma \vdash e : t^*}{\Gamma \vdash *e : t}$$