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: $S : \textbf{String} \to \textbf{int}$ verwaltet :-)

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

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

$$
\begin{array}{l}
\textbf{int}\ \mathsf{getIndex}(\textbf{String}\ w)\ \{ \\
\quad \textbf{if}\ (S\,(w)\ \equiv\ \mathsf{undefined})\ \{ \\
\quad\quad S = S \oplus \{w \mapsto \mathsf{count}\}; \\
\quad\quad \textbf{return}\ \mathsf{count}{+}{+}; \\
\quad \textbf{else}\ \textbf{return}\ S\,(w); \\
\}
\end{array}
$$

Implementierung 2: Partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in$ **String** $\times$ **int** :

  Einfügen: $\mathcal{O}(1)$

  Finden: $\mathcal{O}(n)$ $\Longrightarrow$ zu teuer :-(

- balancierte Bäume :

  Einfügen: $\mathcal{O}(\log(n))$

  Finden: $\mathcal{O}(\log(n))$ $\Longrightarrow$ zu teuer :-(

- Hash Tables :

  Einfügen: $\mathcal{O}(1)$

  Finden: $\mathcal{O}(1)$ ... zumindest im Mittel :-)

... im Beispiel:

- Wir legen ein Feld M von hinreichender Größe m an :-)
- Wir wählen eine Hash-Funktion $H : \textbf{String} \to [0, m-1]$ mit den Eigenschaften:

  $\to$ $H(w)$ ist leicht zu berechnen :-)

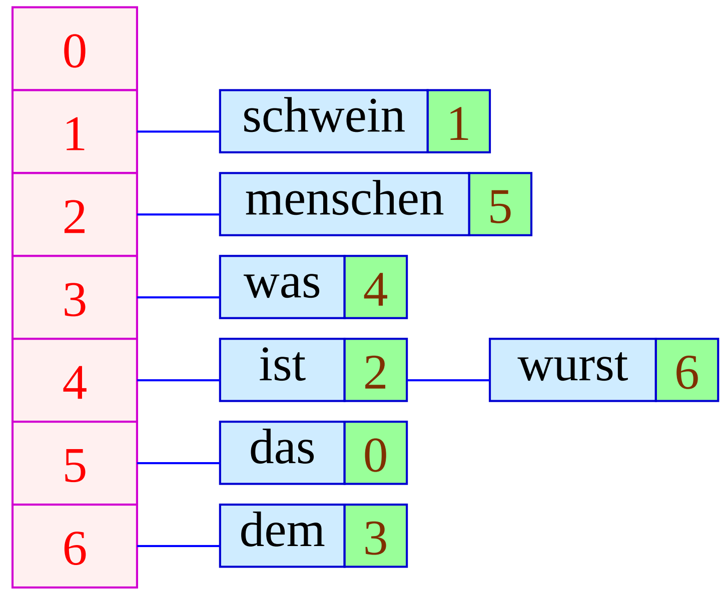
  $\to$ H streut die vorkommenden Wörter gleichmäßig über $[0, m-1]$:-)

  Mögliche Wahlen:

$$
\begin{array}{lll}
H_0(x_0 \ldots x_{r-1}) & = & (x_0 + x_{r-1}) \,\%\, m \\
H_1(x_0 \ldots x_{r-1}) & = & (\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m \\
 & = & (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m \\
 & & \text{für eine Primzahl } p \quad (\text{z.B. } 31 \quad \text{:-)}
\end{array}
$$

- 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

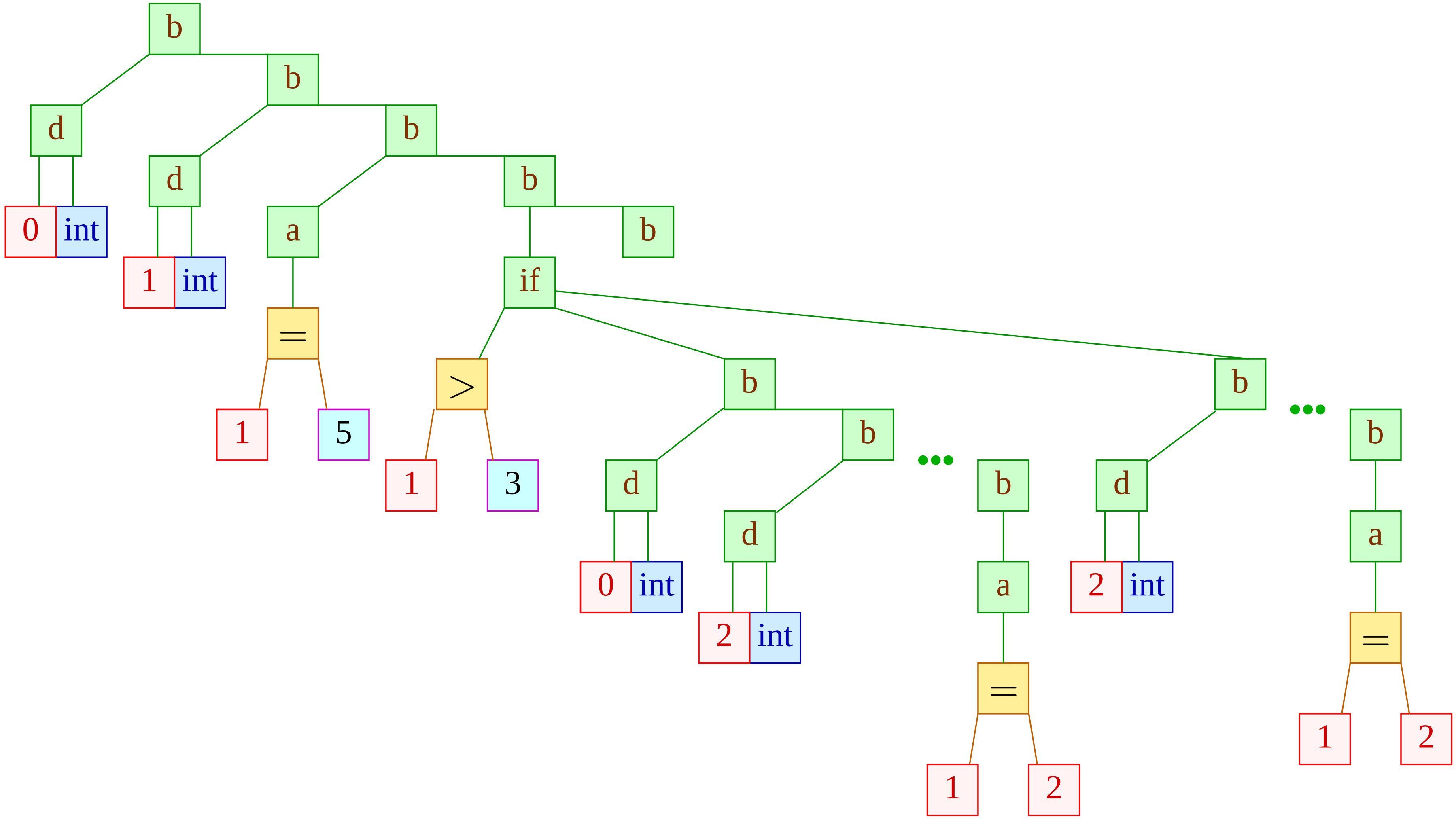
- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
  - $\longrightarrow$ jede Definition vor ihren Benutzungen besucht :-)
  - $\longrightarrow$ die jeweils aktuell sichtbare Definition zuletzt besucht :-)
- Für jeden Bezeichner verwaltet man einen Keller der gültigen Definitionen.
- 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:

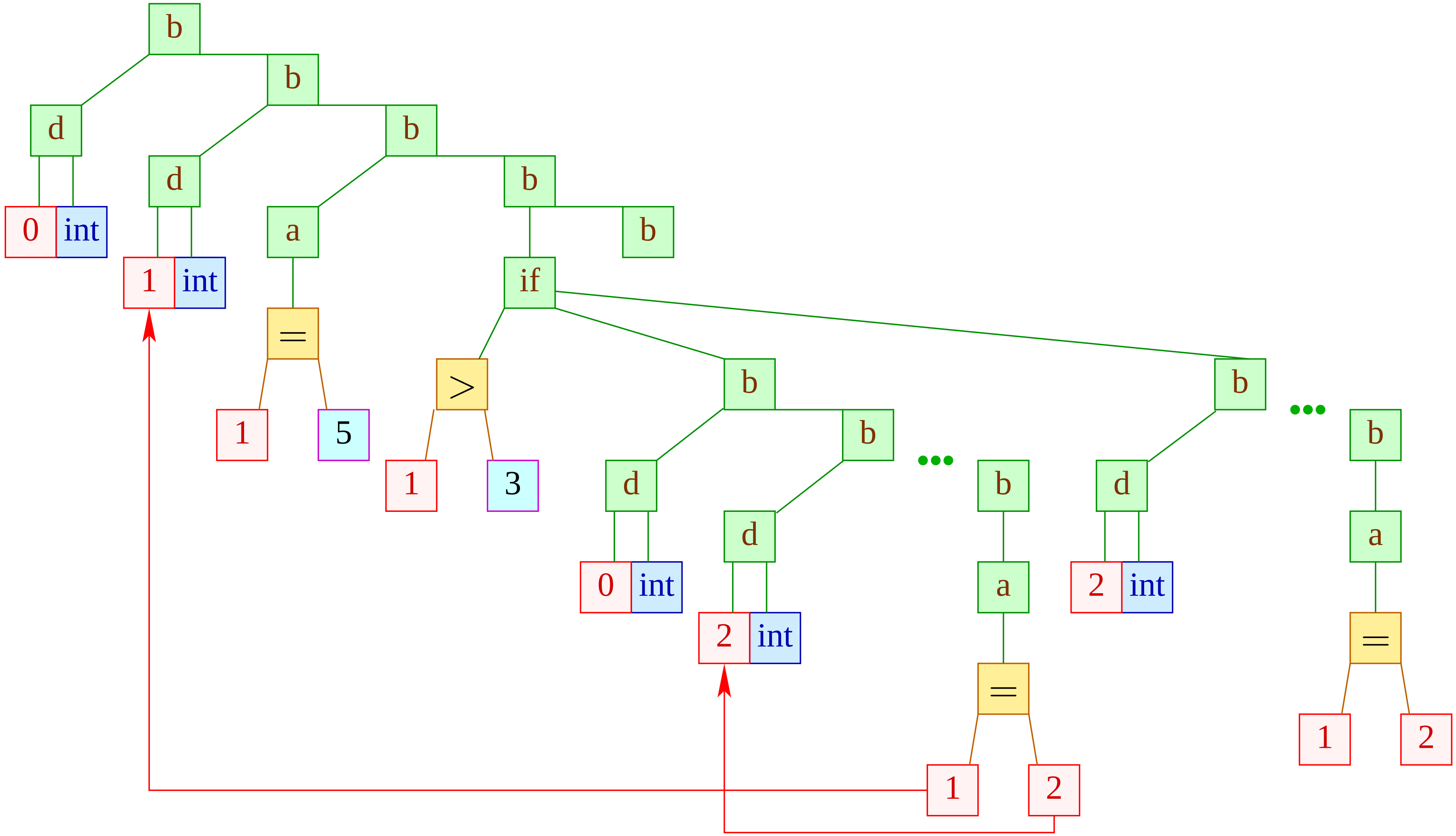
$\{$ **int** $a, b;$
$a = 5;$
if $(a > 3)$ $\{$
int $a, c;$
$a = 3;$
$c = a + 1;$
$b = c;$
$\}$ **else** $\{$
int $c;$
$c = a + 1;$
$b = c;$
$\}$
$b = a + b;$
$\}$

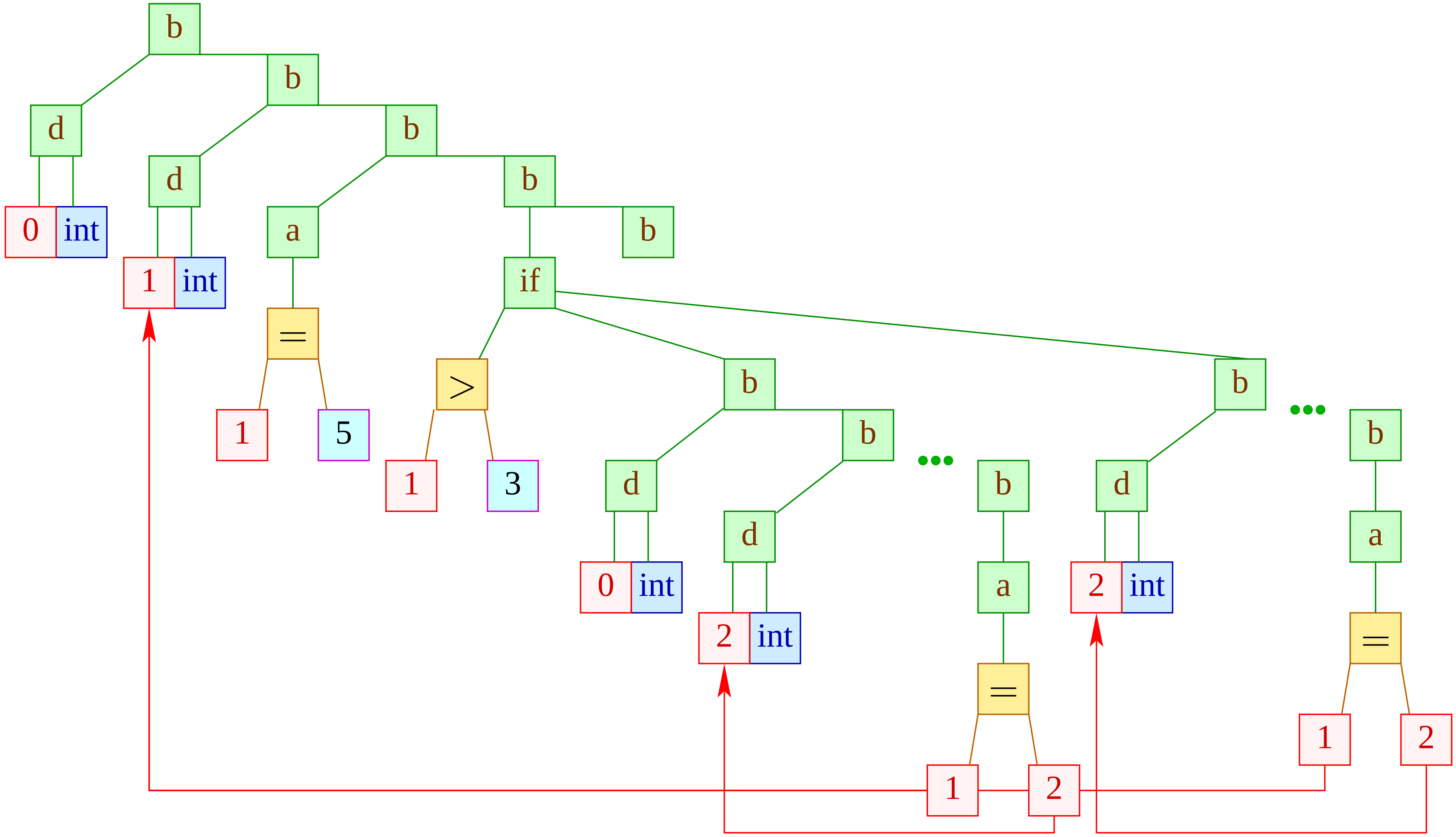
0	a
1	b
2	c

Der zugehörige Syntaxbaum ...



b
b
d
b
d
0
int
b
1
int
a
b
=
if
1
5
>
b
b
1
3
b
d
b
d
b
d
0
int
a
2
int
2
int
a
=
=
1
2
1
2


b
b
d
b
0
int
d
b
1
int
a
b
b
=
if
1
5
>
b
b
1
3
b
d
b
d
0
int
d
a
2
int
2
int
=
a
1
2
=
1
2


b
b
d
b
d
0
int
b
a
1
int
b
if
=
>
b
b
1
5
b
d
1
3
b
b
d
d
a
0
int
2
int
2
int
b
a
=
=
1
2
1
2

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 ...

$$
\begin{array}{clcl}
\textbf{fun} & \mathsf{odd}\ 0 & = & \textbf{false} \\
\mid & \mathsf{odd}\ 1 & = & \textbf{true} \\
\mid & \mathsf{odd}\ x & = & \mathsf{even}\ (x-1) \\
\textbf{and} & \mathsf{even}\ 0 & = & \textbf{true} \\
\mid & \mathsf{even}\ 1 & = & \textbf{false} \\
\mid & \mathsf{even}\ x & = & \mathsf{odd}\ (x-1)
\end{array}
$$

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

  Achtung:
  In C muss zusätzlich eine Größe spezifiziert werden; die Variable muss dann zwischen t und $[n]$ stehen :-(
- Funktionen: $t\ (t_1, \ldots, t_k)$

  Achtung:
  In C muss die Variable zwischen t und $(t_1, \ldots, t_k)$ stehen.

  In SML dagegen würde man diesen Typ anders herum schreiben:
  $t_1 * \ldots * t_k \rightarrow t$:-)

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

(3) Typ-Namen.

Typ-Namen sind nützlich:

- als Abkürzung :-)

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

  $$\textbf{typedef}\ t\ \ x\ ;$$

- 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 $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$

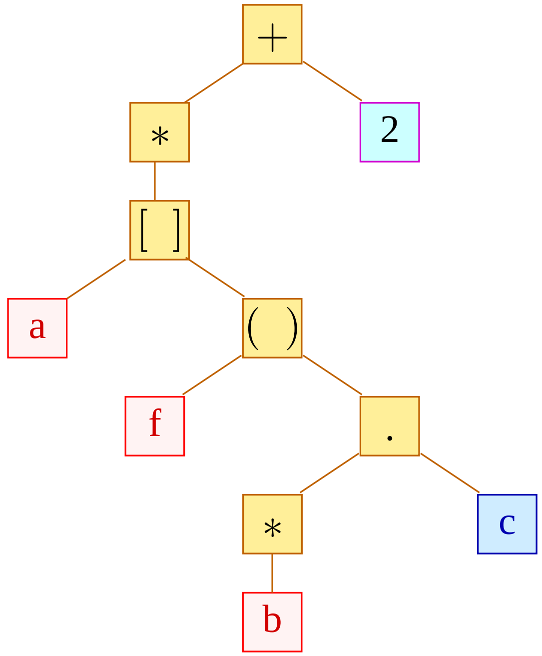
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

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

Betrachte den Ausdruck:

$$*\mathsf{a}[f(\mathsf{b} \to \mathsf{c})] + 2;$$



Idee:

- Traversiere den Syntaxbaum bottom-up.
- Für Bezeichner sagt uns Γ 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 Γ 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:

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

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

Array: $$\frac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Array: $$\frac{\Gamma \vdash e_1 : t[\,] \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Struct: $$\frac{\Gamma \vdash e : \textbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

App: $$\frac{\Gamma \vdash e : t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$$

Op: $$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

Cast: $$\frac{\Gamma \vdash e : t_1 \qquad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2)\ e : t_2}$$