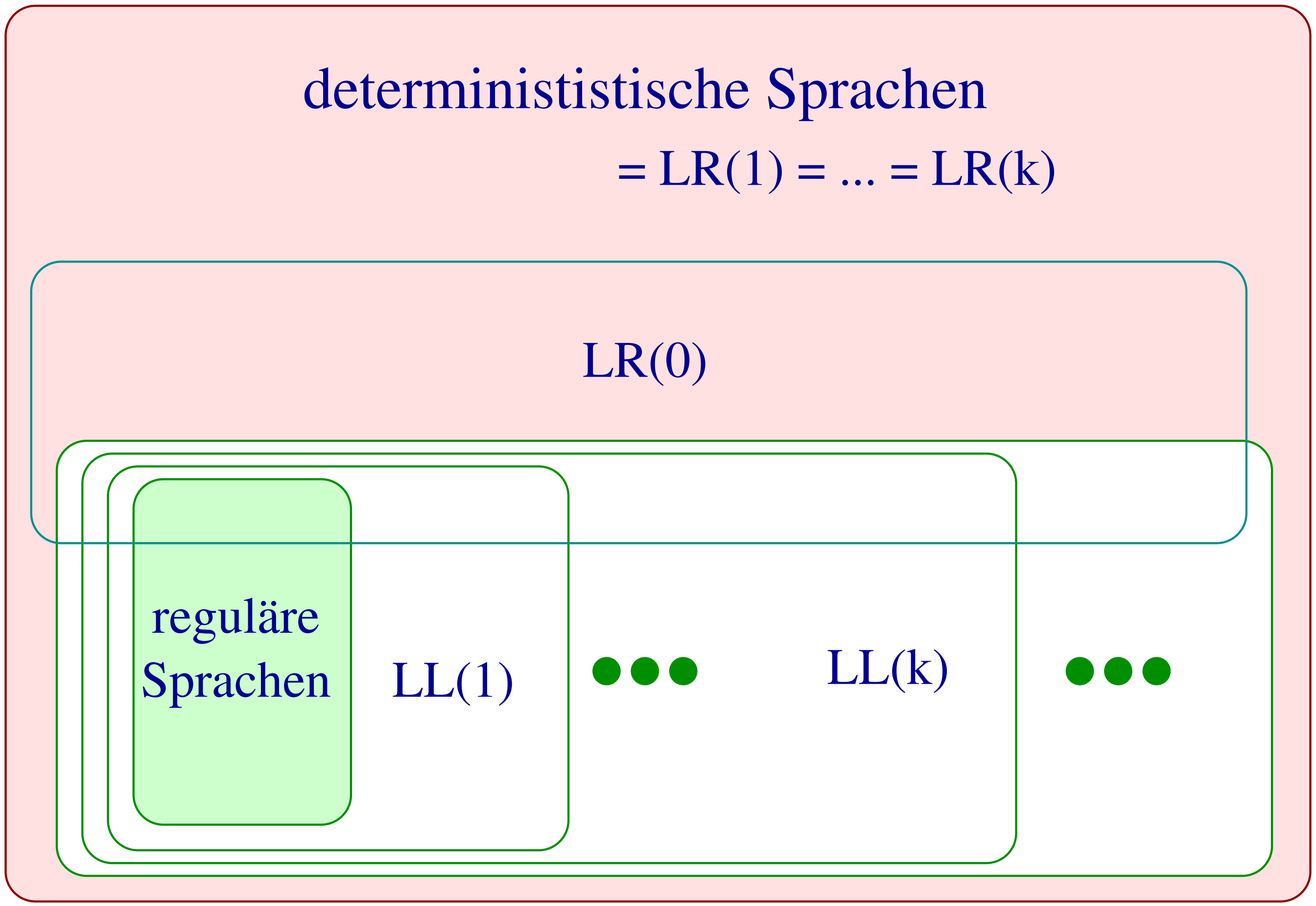
Übersicht über die Sprachklassen:



Diskussion:

- Alle kontextfreien Sprachen, die sich mit einem deterministischen Kellerautomaten parsen lassen, können durch eine LR(1)-Grammatik beschrieben werden.
- Durch LR(0)-Grammatiken lassen sich alle präfixfreien deterministisch kontextfreien Sprachen beschreiben :-)
- Die Sprachklassen zu LL(k)-Grammatiken bilden dagegen eine Hierarchie innerhalb der deterministisch kontextfreien Sprachen.
- Da zu jeder LL(k)-Grammatik eine äquivalente starke LL(k)-Grammatik konstruiert werden kann, sinde letztere nicht in der Übersicht vermerkt.

3 Semantische Analyse

- Lexikalisch und syntaktisch korrekte Programme können trotzdem fehlerhaft sein ;-(
- Einige von diesen Fehlern werden bereits durch die Sprachdefinition ausgeschlossen und müssen vom Compiler überprüft werden :-)
- Weitere Analysen sind erforderlich, um:

  $\longrightarrow$ Bezeichner eindeutig zu machen;

  $\longrightarrow$ die Typen von Variablen zu ermitteln;

  $\longrightarrow$ Möglichkeiten zur Programm-Optimierung zu finden.

3.1 Symbol-Tabellen

Beispiel:

```
void foo() {
    int A;
    void fee() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    fee();
    write(A);
}
```

Diskussion:

- Innerhalb des Rumpfs von fee wird die Definition von A durch die lokale Definition verdeckt :-)
- Für die Code-Erzeugung benötigen wir für jede Benutzung eines Bezeichners die zugehörige Definitionsstelle.
- Statische Bindung bedeutet, dass dass die Definition eines Namens A an allen Programmpunkten innerhalb ihres gesamten Blocks gültig ist.
- Sichtbar ist sie aber nur außerhalb derjenigen Teilbereiche, in an denen eine weitere Definition von A gültig ist :-)

... im Beispiel:

```
void foo() {
    int A;
    void fee() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    fee();
    write(A);
}
```

Kompliziertere Regeln der Sichtbarkeit gibt es in objektorientierten Programmiersprachen wie Java ...

Beispiel:

```
public class Foo {
    protected int x = 17;
    protected int y = 5;
    private int z = 42;
    public int b() { return 1; }
}
class Fee extends Foo {
    protected double y = .5;
    public int b(int a) { return a; }
}
```

Diskussion:

- **private** Members sind nur innerhalb der aktuellen Klasse gültig :-)
- **protected** Members sind innerhalb der Klasse, in den Unterklassen sowie innerhalb des gesamten package gültig :-)
- Methoden b gleichen Namens sind stets verschieden, wenn ihre Argument-Typen verschieden sind !!!
- Bei Aufrufen einer Methode wird dynamisch entschieden, welche Definition gemeint ist ...

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

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

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

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

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:

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

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

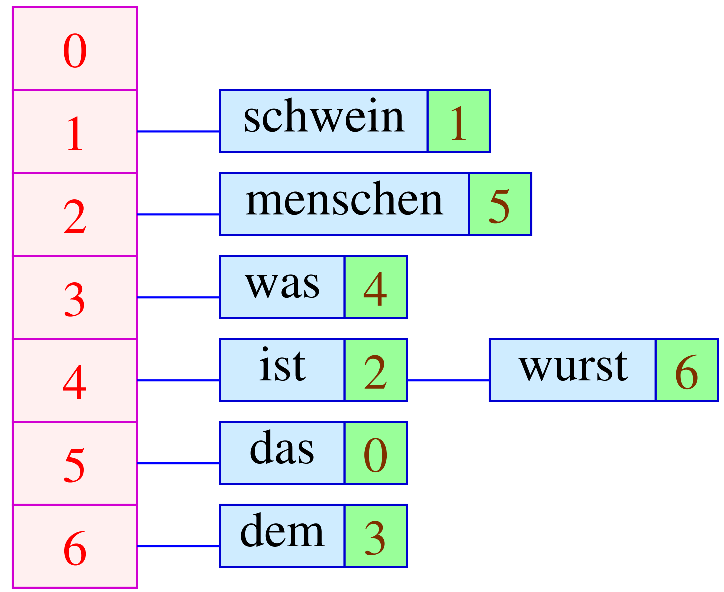
Mögliche Wahlen:

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

:-)

- 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

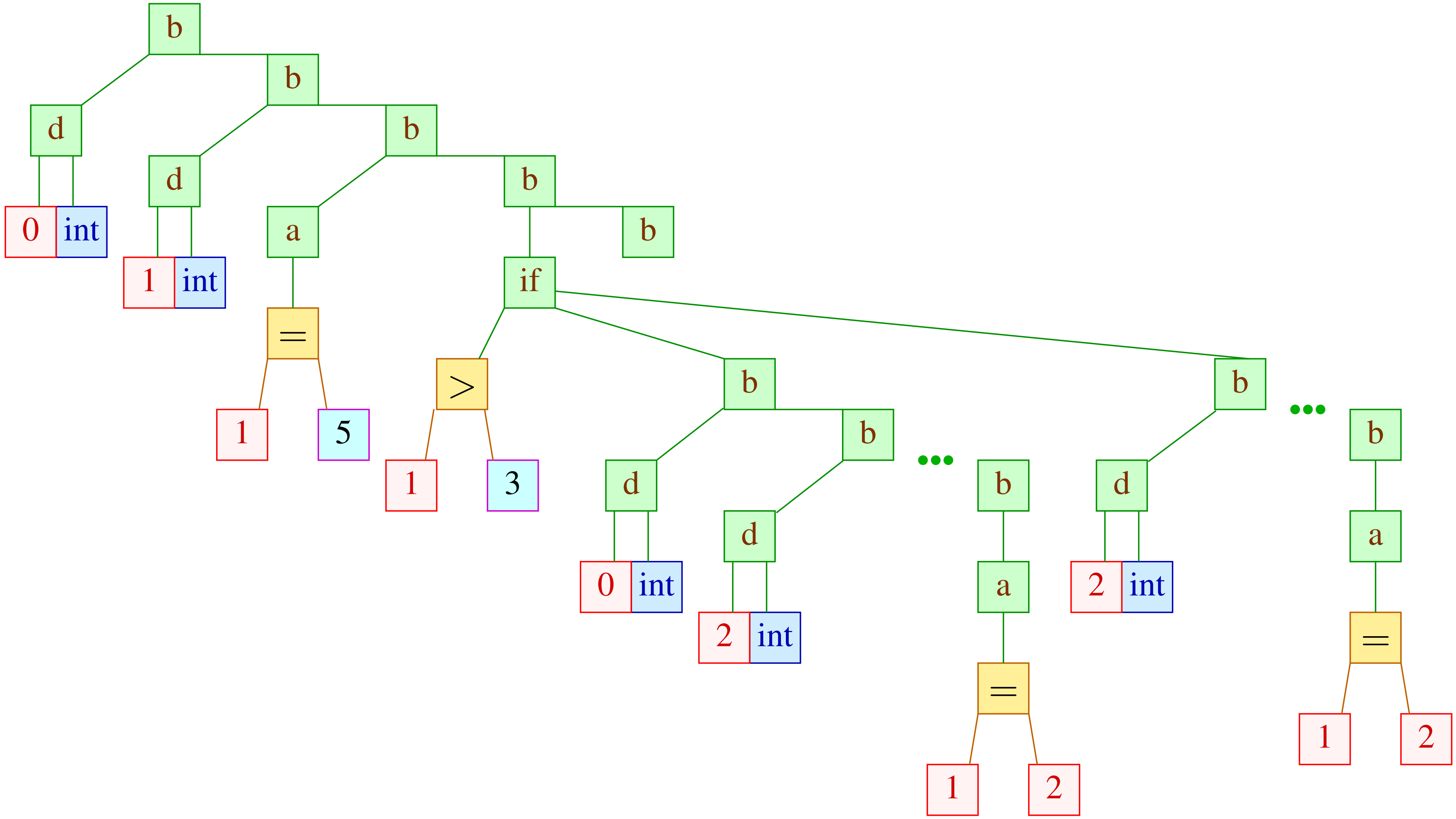
- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
  - → jede Definition vor ihren Benutzungen besucht :-)
  - → die jeweils aktuell sichtbare Definition zuletzt besucht :-)
- Für jeden Bezeichner verwaltet man einen Keller der gültigen Definitionen.
- Trifft man bei der Durchmusterung 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;
  }
  b = a + b;
}
```

0	a
1	b
2	c

Der zugehörige Syntaxbaum ...


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