Tutorium 06: Prolog

Paul Brinkmeier

06. Dezember 2022

Tutorium Programmierparadigmen am KIT

Übungsblätter

Wiederholung

Cheatsheet: Lambda-Kalkül/Basics

- Terme t: Variable ($\mathtt{x}$), Funktion ($\lambda \mathtt{x}.\, t$), Anwendung ($t\ t$)
- α-*Äquivalenz*: Gleiche Struktur
- η-*Äquivalenz*: Unterversorgung
- *Freie Variablen*, *Substitution*, *RedEx*
- β-*Reduktion*:
  $(\lambda \mathtt{p}.\, b)\ t \Rightarrow b\,[\mathtt{p} \to t]$

Cheatsheet: Lambda-Kalkül/Fortgeschritten

- Auswertungsstrategien (von lässig nach streng):
  - *Volle β-Reduktion*
  - *Normalreihenfolge*
  - *Call-by-Name*
  - *Call-by-Value*
- Datenstrukturen:
  - *Church-Booleans*
  - *Church-Zahlen*
  - *Church-Listen*
- Rekursion durch *Y-Kombinator*

Übungsblatt 4

Y-Kombinator

Globale Definitionen im λ-Kalkül

$$\mathit{fibAcc} = \lambda \mathtt{a}.\, \lambda \mathtt{b}.\, \lambda \mathtt{n}.\, \mathit{isZero}\ \mathtt{n}\ \mathtt{a}\ (\mathit{fibAcc}\ \mathtt{b}\ (\mathit{add}\ \mathtt{a}\ \mathtt{b})\ (\mathit{pred}\ \mathtt{n}))$$

```
fibAcc a b n = if n == 0 then a else fibAcc b (a+b) (n-1)
```

- Definition von *fibAcc* ist nicht Teil des λ-Kalküls
  (λ enthält *nur* Variablen, Funktionsanwendung, Abstraktion)
- $\Longrightarrow$ *fibAcc* „sieht“ sich selbst nicht
  „*fibAcc* ist frei in $\lambda \mathtt{a}.\, \lambda \mathtt{b}.\, \ldots$“

$$\underbrace{\mathit{fibAcc}\ c_0\ c_1\ c_7 \overset{*}{\Longrightarrow}}_{\text{Aus Definition}} \underbrace{\mathit{fibAcc}\ c_1\ c_1\ c_6 \not\Longrightarrow}_{\textit{fibAcc} \text{ frei}}$$

Rekursive Funktionen

- Damit *fibAcc* nicht frei ist, müssen wir es *binden*.
  Im λ-Kalkül gibt es dafür nur ein mögliches Konstrukt:
- Ein λ, das *fibAcc* als Parameter nimmt!

$$\lambda \mathtt{a}.\, \lambda \mathtt{b}.\, \lambda \mathtt{n}.\, \mathit{isZero}\ \mathtt{n}\ \mathtt{a}\ (\mathit{fibAcc}\ \mathtt{b}\ (\mathit{add}\ \mathtt{a}\ \mathtt{b})\ (\mathit{pred}\ \mathtt{n}))$$

$$\rightsquigarrow \lambda \mathtt{fibAcc}.\, \lambda \mathtt{a}.\, \lambda \mathtt{b}.\, \lambda \mathtt{n}.\, \mathit{isZero}\ \mathtt{n}\ \mathtt{a}\ (\mathtt{fibAcc}\ \mathtt{b}\ (\mathit{add}\ \mathtt{a}\ \mathtt{b})\ (\mathit{pred}\ \mathtt{n}))$$

- Um auf unsere ursprünglich gewollte Funktion zu kommen, wenden wir Y an:

$$Y = \lambda \mathtt{f}.\, (\lambda \mathtt{x}.\, \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda \mathtt{x}.\, \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))$$

$$\mathit{fibAcc2} = Y\ (\lambda \mathtt{fibAcc}.\, \lambda \mathtt{a}.\, \lambda \mathtt{b}.\, \ldots \mathtt{fibAcc} \ldots)$$

Einführung in Prolog

Prolog — Umgebung

- Prolog ist eine Programmiersprache, wenn auch eine seltsame
- $\rightsquigarrow$ gut wird man durch Übung
- Zum Üben:
  - SWI-Prolog — gängige Prolog-Umgebung
  - SWISH — SWI-Prolog Web-IDE zum Testen
  - VIPR, VIPER — PSE-Tools des IPD, auf der Seite der Übung verlinkt

Prolog — Regelsysteme als Programmiersprache

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

mother(inge, emil).
mother(inge, petra).
father(emil, kunibert).
```

`?- grandparent(inge, kunibert).` $\rightsquigarrow$ `yes.`

Prolog — Regelsysteme als Programmiersprache

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

mother(inge, emil).
mother(inge, petra).
father(emil, kunibert).
```

$$\dfrac{\dfrac{\dfrac{}{\texttt{mother(inge, emil)}}}{\texttt{parent(inge, emil)}} \quad \dfrac{\dfrac{}{\texttt{father(emil, kunibert)}}}{\texttt{parent(emil, kunibert)}}}{\texttt{grandparent(inge, kunibert)}}$$

Funktoren

```
a(b, c, d).
defg.
bintree(bintree(1, 2), bintree(3, bintree(4, 5))).
list(cons(1, cons(2, cons(3, nil)))).
'Abcd'('X', 'Y', 'Z').
```

- Funktor $\approx$ Name + Liste von Prolog-Ausdrücken
- Liste leer $\rightsquigarrow$ „Atom“
- Name wird immer klein geschrieben
  - Großbuchstaben: bspw. `'List'`
- Auch mathematische Ausdrücke sind Funktoren:
  `17 + 25` $\approx$ `'+'(17, 25)`

Variablen

```
?- X = pumpkin.
?- Y = honey_bunny.
?- Z = vincent.
?- [A, B, C] = [1, 2, 3].
?- f(L, rechts) = f(links, R)
```

- Variablen werden immer groß geschrieben
- = ist nicht Zuweisung, sondern Unifikation
- Unifikation $\approx$ (formales) Pattern-Matching

Ziele

```
main :-
   [A, B, C] = [1, 2, 3],
   max(A, C, X),
   Y is X * 2,
   Y > 0,
   not(Y > 10),
   !.
```

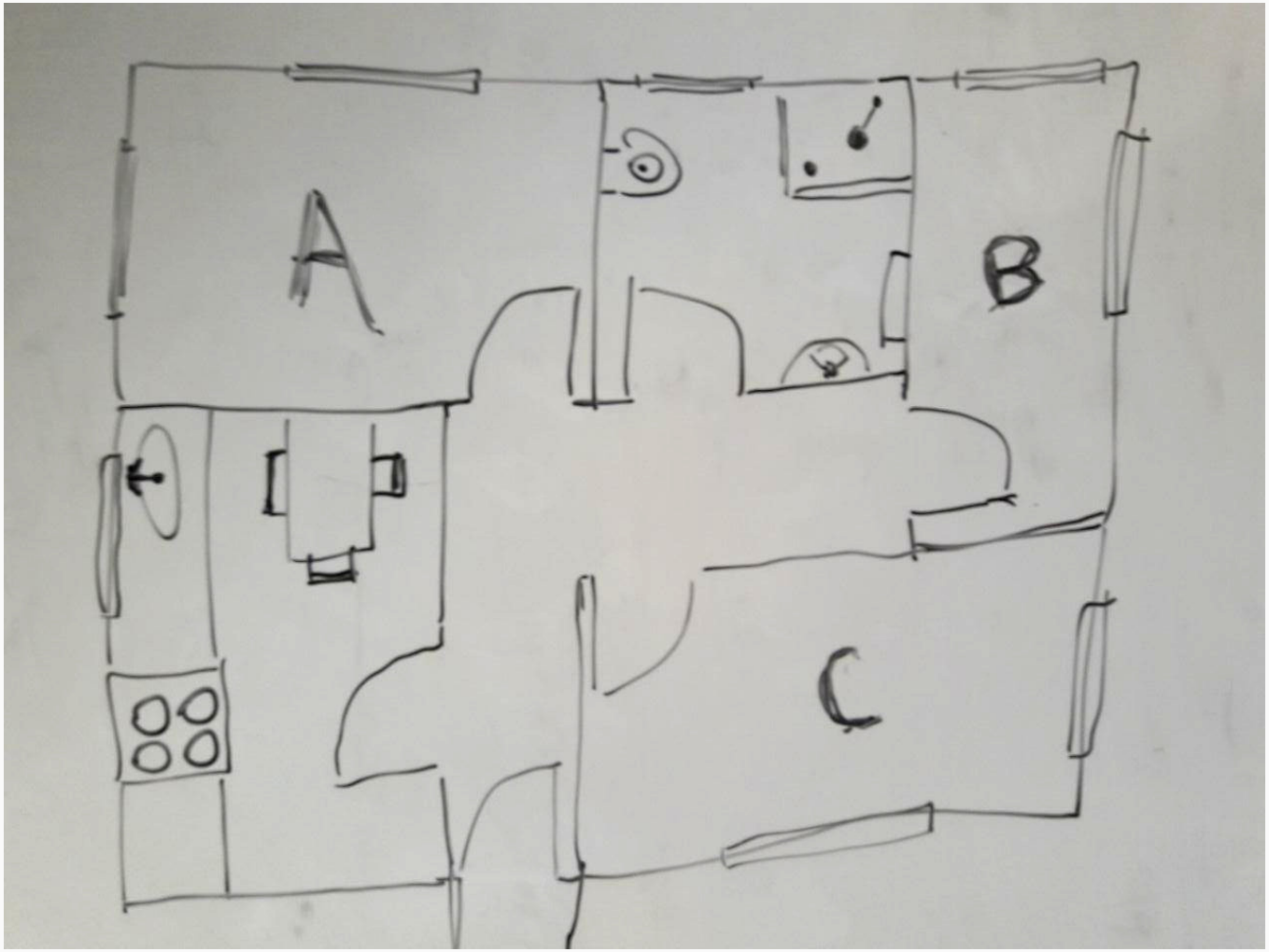
- Funktionsaufruf $\approx$ „Zielerfüllung“ in Prolog
- Mögliche Ziele:
  - Unifikationsziel
  - *Funktorziel*
  - Arithmetische Zuweisung
  - Arithmetischer Vergleich
  - Nicht-Erfüllung
  - Cut

Programme

- Prolog-„Programme“ $\approx$ Datenbanken
- Ausführung $\approx$ Abfrage in der Datenbank
- Datenbank-Inhalt: Regeln, bestehend aus:
  - Regelkopf — Ein Funktor $\rightsquigarrow$ kann auch Atom sein
  - Teilziele — Liste von Zielen, um diese Regel zu erfüllen
  - Keine Teilziele $\rightsquigarrow$ Fakt

Prolog-Aufgaben

Mathematiker-WG



Mathematiker-WG

- Alice, Bob und Carl ziehen in eine WG
- Die drei sind Mathematiker;
  jeder will eine eigene Zahl von 1 bis 7 für sein Zimmer
- Die Summe der Zahlen soll 12 sein
- Alice mag keine ungeraden Zahlen

Findet alle 14 möglichen Kombinationen, die Zimmer zu nummerieren.

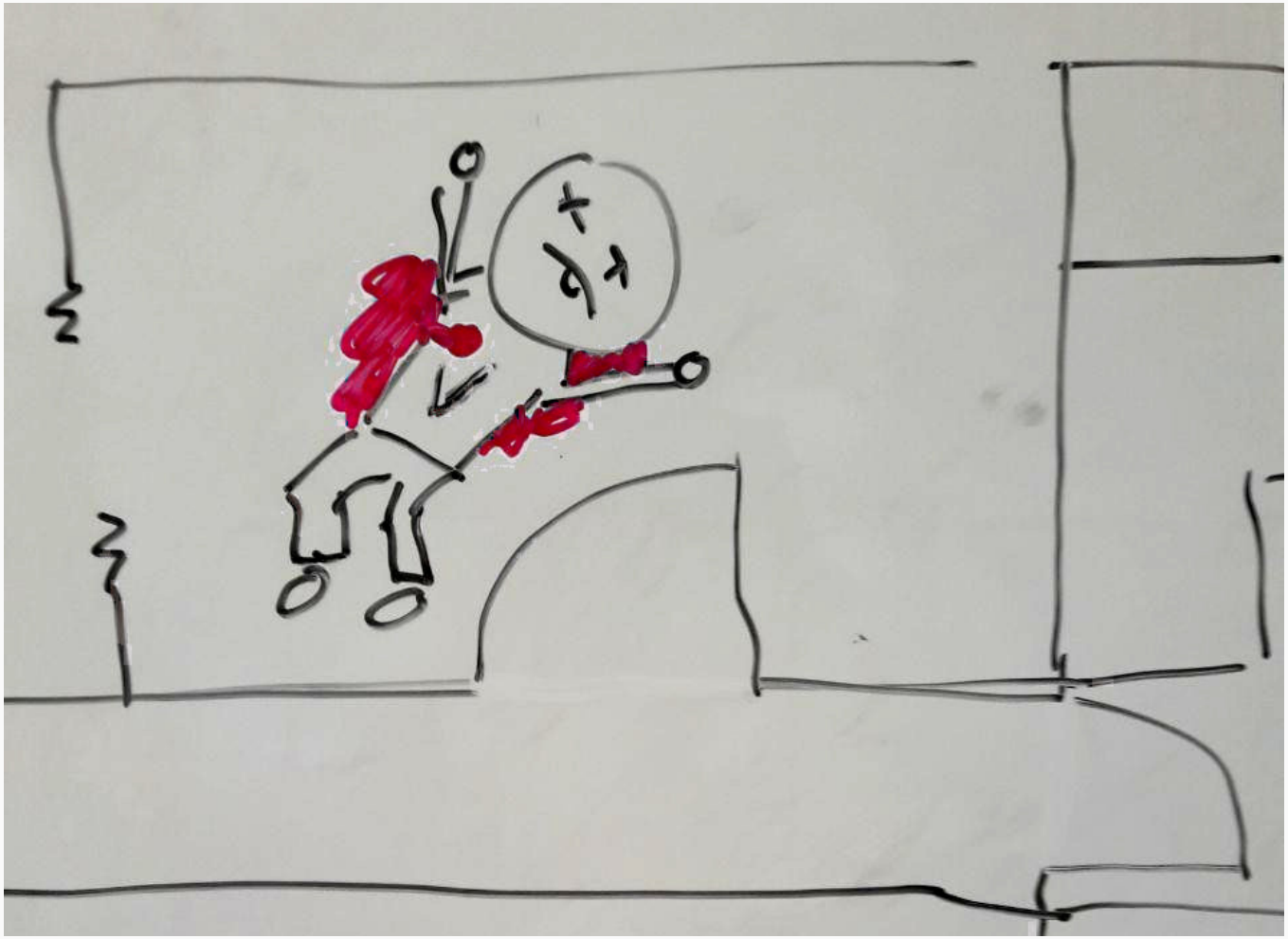
Mathematiker-WG

```
% mathematiker_wg.pl

alice(2).
alice(4).
alice(6).
...

nummerierung(A, B, C) :-
  alice(A),
  bob(B),
  carl(C),
  ...
  12 =:= A + B + C.
```

Detektivrätsel



Detektivrätsel

Im Fall des Mordes an ihrem Nachbarn Victor sind nun Alice, Bob und Carl die einzigen Verdächtigen und Zeugen.

- Alice:
  - Bob war mit dem Opfer befreundet.
  - Carl und das Opfer waren verfeindet.
- Bob:
  - Ich war überhaupt nicht daheim!
  - Ich kenne den garnicht!
- Carl:
  - Ich bin unschuldig!
  - Wir waren zum Zeitpunkt der Tat alle in der WG.

Detektivrätsel

```
% detektiv.pl

aussage(alice, freund(bob)).
aussage(alice, feind(carl)).
...

% Widersprüche

widerspruch(freund(X), feind(X)).
...
```

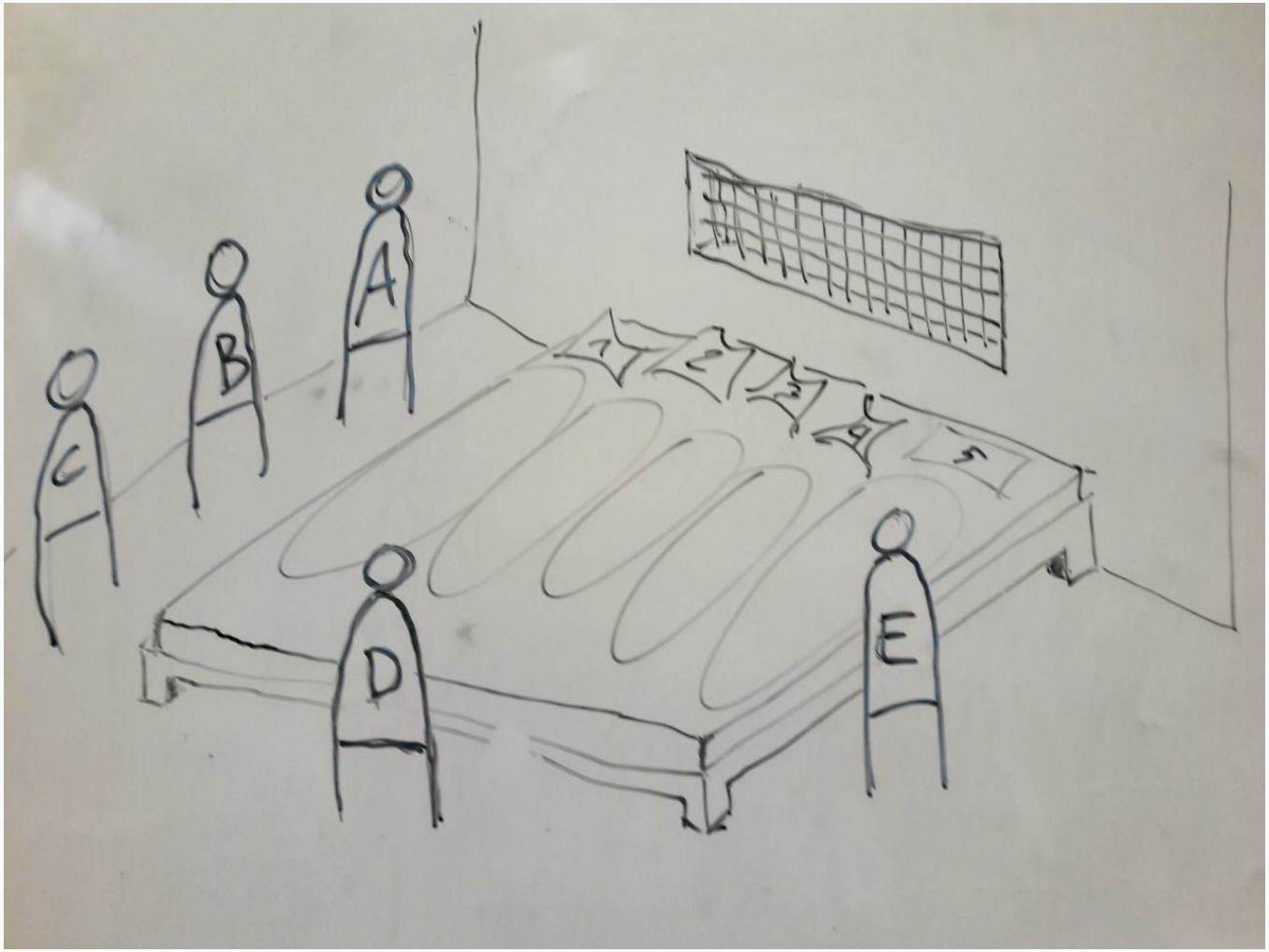
Detektivrätsel

```
taeter(T) :-
  select(T, [alice, bob, carl], Rest),
  not(inkonsistent(Rest)).

inkonsistent(Zeugen) :- ...
```

- `select(X, Xs, Ys)` generiert Elemente `X` aus `Xs` mit Restlisten `Ys`.
- Implementiert: `inkonsistent/1`
  Überprüft Aussagen von Zeugen paarweise auf Widerspruch

Schlafplätze im Gefängnis



Dinesman's multiple-dwelling problem

Bob kommt nun ins Gefängnis. Aaron, Bob, Connor, David und Edison müssen sich zu fünft ein sehr breites Bett teilen.

- Aaron will nicht am rechten Ende liegen
- Bob will nicht am linken Ende liegen
- Connor will an keinem der beiden Enden liegen
- David will weiter rechts liegen als Bob
- Connor schnarcht sehr laut;
  Bob und Edison sind sehr geräuschempfindlich
  - $\rightsquigarrow$ Bob will nicht direkt neben Connor liegen
  - $\rightsquigarrow$ Edison will nicht direkt neben Connor liegen

Wie können die 5 Schlafplätze verteilt werden?

Schlafplätze im Gefängnis

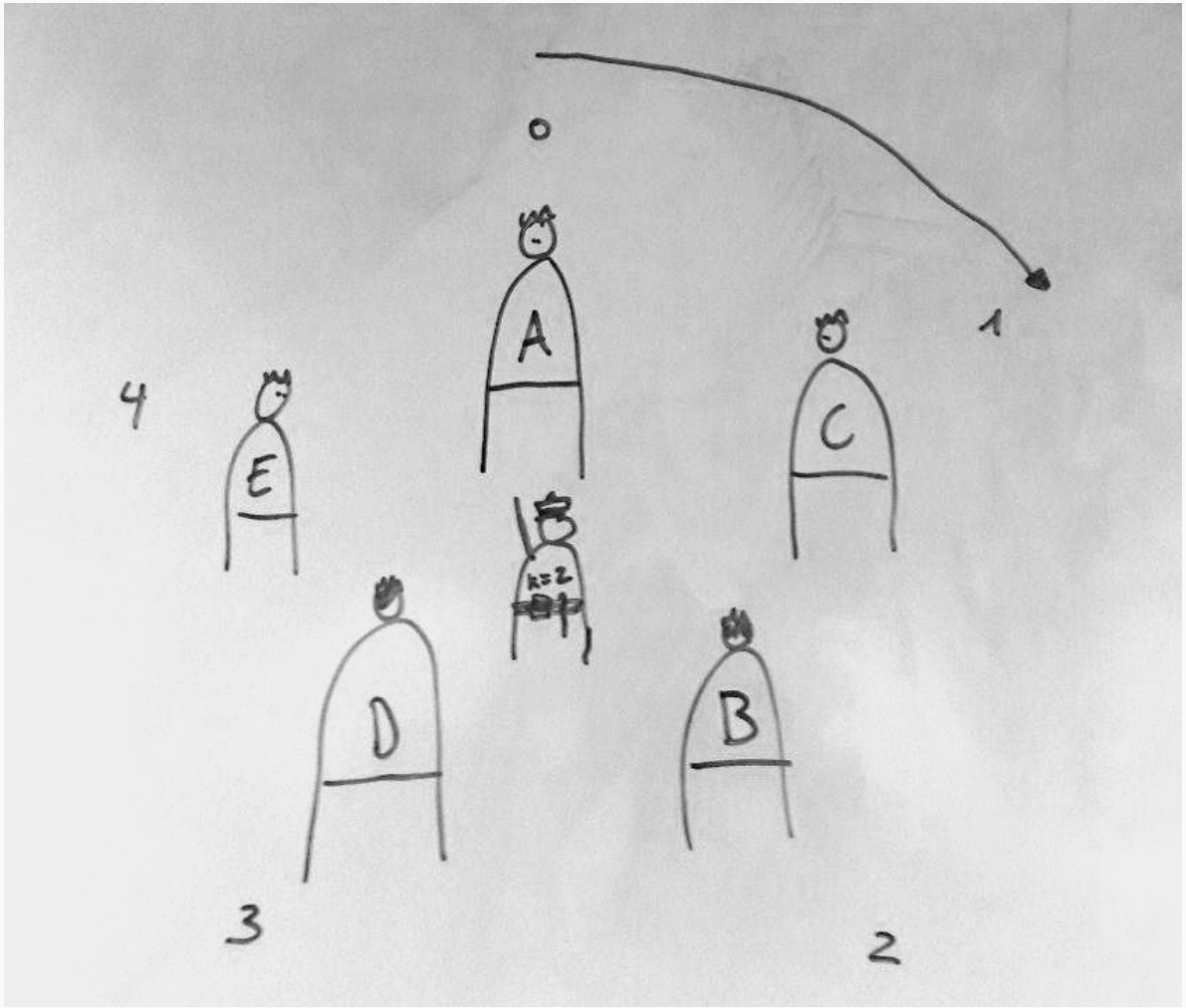
```
% schlafplaetze.pl

bett(X) :- member(X, [1, 2, 3, 4, 5]).

schlafplaetze(A, B, C, D, E) :-
  bett(A), bett(B), bett(C), bett(D), bett(E),
  distinct([A, B, C, D, E]),
  % weitere Tests
```

- Fügt weitere benötigte Tests ein
- Implementiert:
  - `distinct/1` prüft Listenelemente auf paarweise Ungleichheit
  - `adjacent/2` prüft, ob $|A - B| = 1$

Putzdienst



Putzdienst

- Aaron, Bob, Connor, David und Edison sollen 4 Einheiten Putzdienst übernehmen
- Da sie sich nicht einigen können, wer aussetzen darf, wendet ein Wärter folgendes Vorgehen an:
  - Die fünf werden im Kreis aufgestellt
  - Der Wärter stellt sich in die Mitte
  - Beginnend bei 12 Uhr dreht er sich im Uhrzeigersinn und teilt jeden zweiten Insassen zum Putzdienst ein
    - D.h. es wird immer ein Insasse übersprungen

An welcher Stelle muss Bob stehen, um nicht putzen zu müssen?

Putzdienst

- Aaron, Bob, Connor, David und Edison sollen 4 Einheiten Putzdienst übernehmen
- Da sie sich nicht einigen können, wer aussetzen darf, wendet ein Wärter folgendes Vorgehen an:
  - Die fünf werden im Kreis aufgestellt
  - Der Wärter stellt sich in die Mitte
  - Beginnend bei 12 Uhr dreht er sich im Uhrzeigersinn und teilt jeden zweiten Insassen zum Putzdienst ein
    - D.h. es wird immer ein Insasse übersprungen

An welcher Stelle muss Bob stehen, um nicht putzen zu müssen?

An welcher Stelle muss Bob bei 41 Insassen stehen, wenn immer jeder *dritte* Insasse eingeteilt wird?

Putzdienst

```
% putzdienst.pl

% Bspw.
% ?- keinPutzdienstFuer([a, b, c, d, e], 2, X)
keinPutzdienstFuer(L, K, X) :-
  helper(L, K, K, X).

helper([X], _C, _K, X) :- !.
helper([H|T], 1, K, X) :- ...
...
```

- Weitere Fälle für `helper/4`:
  - `C = 1` $\rightsquigarrow$ Element entfernen
  - Ansonsten: Element hinten wieder anhängen

Quellen der Aufgaben

Zum Nachlesen und Vergleichen mit Lösungen in anderen Programmiersprachen:

- WG — Rosetta Code: Department Numbers
- Detektiv — github.com/Anniepoo/prolog-examples
- Schlafplätze — SICP, S. 418
- Putzdienst — Rosetta Code: Josephus problem