Informatik 1

Assignment Project Exam Help

https://tutorcs.com

06 – Polymorphe Signaturen, Polymorphe Paare und Listen

WeChat: cstutorcs

Winter 2020/21

Torsten Grust
Universität Tübingen, Germany

1 “Flexible” Funktionen

Nochmals ein genauer Blick auf die Identitätsfunktion `id`:

```
; Identität: liefere Argument x zurück
(: id (□₁ -> □₂))
(define id
  (lambda (x)
    x))
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- Wie sollten die Signaturen $\square_1$ und $\square_2$ lauten?
  1. `id` liefert sein Argument `x` *unverändert* zurück.
     Also scheint $\square_1 \equiv \square_2$ sinnvoll.
  2. `id` funktioniert auf *beliebigen* Werten.
     Also $\square_1 \equiv$ `any`.
  3. Mit 1. + 2. also: `(: id (any -> any))`? 😒 ***Nein!***

2 ¦ (Parametrisch) Polymorphe Funktionen

(Parametrisch) Polymorphe Funktionen arbeiten *unabhängig* von den Signaturen ihrer Argumente (*polýmorphos*: vielgestaltig).

- **Signaturvariablen** (%a, %b, %c, ...) drücken aus, dass...
  1. eine Signatur **beliebig** sein kann und
  2. zwei (oder mehr) Signaturen **identisch** sein müssen.

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

Beispiel Identitätsfunktion id:

```
          ┌─ = ─┐ ┄┄┄┄ Signaturen von Argument und Resultat
          ↓     ↓       beliebig aber identisch
(: id (%a -> %a))
(define id
  (lambda (x)
    x))
```

Mehr polymorphe Funktionen

```
; konstante Funktion: ignoriere zweites Argument
(: const (%a %b -> %a))
(define const
  (lambda (x y) x))

; Projektion: ein Argument auswählen
(: proj ((one-of 1 2) %a %b -> (mixed %a %b)))
(define proj
  (lambda (i x1 x2)
    (cond
      ((= i 1) x1)
      ((= i 2) x2))))
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

Parametrisch polymorphe Funktionen “wissen rein gar nichts” über ihre Argumente mit Signatur %a, %b, … und könne diese **nur** reproduzieren oder an andere polymorphe Funktionen weiterreichen.

Polymorphe Signaturen

Eine **polymorphe Signatur** steht für alle Signaturen, in denen die Signaturvariablen %a, %b, ... *konsistent* durch konkrete Signaturen ersetzt werden.

Beispiel:

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- Wenn *f* die polymorphe Signatur (%a number %b -> %a) besitzt, kann *f* als Funktion mit folgenden Signaturen agieren:

```
(string  number boolean -> string)          { %a ≡ boolean
(boolean number natural -> boolean) ..... {
(string  number string  -> string)          { %b ≡ natural
(number  number number  -> number)
        ⋮
```

3 ¦ Paare von Werten — Drei Mal (fast) das Gleiche…

Drei Records, die jeweils *Paare von Werten* darstellen:

❶

temp	36.9
unit	"°C"

❷

lat	-50.0
long	30.0

❸

num	37
prime?	#t

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- Jeweils **ein zweisteiliger Konstruktor** und jeweils **zwei Selektoren.**
- Die drei Record-Definitionen unterscheiden sich nur durch die Signaturen der Konstruktoren und Selektoren.[1]
- 💡 Ist der dreifache Aufwand vermeidbar, wenn **Konstruktor und Selektoren polymorphe Signaturen** verwenden...?

[1] OK, die Namen der Selektoren unterscheiden sich auch. Wir könnten uns auf `first` und `second` einigen.

Polymorphe Records (hier: Polymorphes Paar)

```
; Ein polymorphes Paar (pair) besteht aus
; - erster Komponente (first)
; - zweiter Komponente (rest)
; wobei die Komponenten jeweils beliebige Werte sind:

(define-record-procedures-parametric pair [pair-of]      ❶
  make-pair                                               ❷ Konstruktor
  pair?                                                   ❸ Prädikat
  (first                                                  ❹ Selektoren
   rest))
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- ❶ Hier `[]` definiert die Spezialform die **parametrisierte Signatur** (pair-of t_1 t_2) für Paare, deren erste/zweite Komponente die Signatur t_1/t_2 besitzen.

Nochmal: Drei Paare von Werten 😀

❶

first	36.9
rest	"°C"

(pair-of real string)

❷

first	-50.0
rest	30.0

(pair-of real real)

❸

first	37
rest	#t

(pair-of natural boolean)

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- Offensichtlich müssen Konstruktor make-pair und Selektoren first und rest **polymorph** sein:

```
(first (make-pair 36.9 "°C")  ⇝ 36.9
(first (make-pair 37 #t)      ⇝ 37
(rest  (make-pair 36.9 "°C")) ⇝ "°C"
(rest  (make-pair 37 #t))     ⇝ #t
```

↖ ((pair-of natural boolean) -> boolean)

Polymorphe Paare (≡ polymorpher Konstruktor und Selektoren)

Record-Definition für **polymorphe Paare** (pair-of t_1 t_2), komplett mit Signaturen für Konstruktor und Selektoren:

```
; Ein polymorphes Paar (pair) besteht aus
; - erster Komponente (first)
; - zweiter Komponente (rest)
; wobei die Komponenten jeweils beliebige Werte sind:
(: make-pair (%a %b -> (pair-of %a %b)))          ; Konstruktor
(: pair?     (any -> boolean))                    ; Prädikat
(: first     ((pair-of %a %b) -> %a))             ; Selektoren
(: rest      ((pair-of %a %b) -> %b))             ;
(define-record-procedures-parametric pair pair-of
  make-pair
  pair?
  (first
   rest))
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- (Zusätzlich definiert: Signatur pair ≡ (pair-of any any), repräsentiert beliebige Paare.)

Eine **Liste (list-of *t*) von Elementen der Signatur** *t* ist

- leer ❶ *oder*
- ein Paar, bestehend aus
  - einem **Listenkopf** ❷ der Signatur *t* und
  - einer **Restliste** ❸ der Signatur (list-of *t*):

Assignment Project Exam Help

https://tutorcs.com

```
; (list-of t): Listen von Werten der Signatur t
(define list-of
  (lambda (t)
    (signature (mixed empty-list         ; ❶
                      (pair-of t (list-of t))))))
                               ; ❷: t   ❸: (list-of t)
```

WeChat: cstutorcs

Die **Restliste** ist **selbst wieder eine** (kürzere) **Liste**, siehe (◄---►): Listen sind **rekursive** Datenstrukturen.

Listen: Leere Liste

Leere Listen (0 Elemente) besitzen die Signatur `empty-list`.

- Racket kennt vordefinierte Werte und Funktionen für Operationen auf leeren Listen:

Assignment Project Exam Help

```
(: empty empty-list)          ; die leere Liste (Literal)
(: empty? (any -> boolean))   ; ist dies eine leere Liste?
```

https://tutorcs.com

WeChat: cstutorcs

- Also:

```
empty                                ⇝  #<empty-list>
(empty? empty)                       ⇝  #t
(empty? (make-pair 42 empty))        ⇝  #f
(empty? 0)                           ⇝  #f
```

Zusammenfassung: Operationen auf Listen

- **Listenkonstruktion:**

```
(: empty empty-list)                                    ; leere Liste
(: make-pair (%a (list-of %a) -> (list-of %a)))
;  ↑ konstruiere nicht-leere Liste aus Kopf und Restliste
```

Assignment Project Exam Help

- **Prädikate:**

https://tutorcs.com

```
(: empty? (any -> boolean))                             ; leere Liste?
(: pair?  (any -> boolean))                             ; nicht-leere Liste?
```

WeChat: cstutorcs

- **Selektoren:**

```
(: first ((list-of %a) -> %a))                          ; Kopfelement
(: rest  ((list-of %a) -> (list-of %a)))                ; Restliste
```

5 ¦ Visualisierung von Listen

Listen sind *die* Container-Datenstruktur im Functional Programming. Wie wollen wir Listen visualisieren?

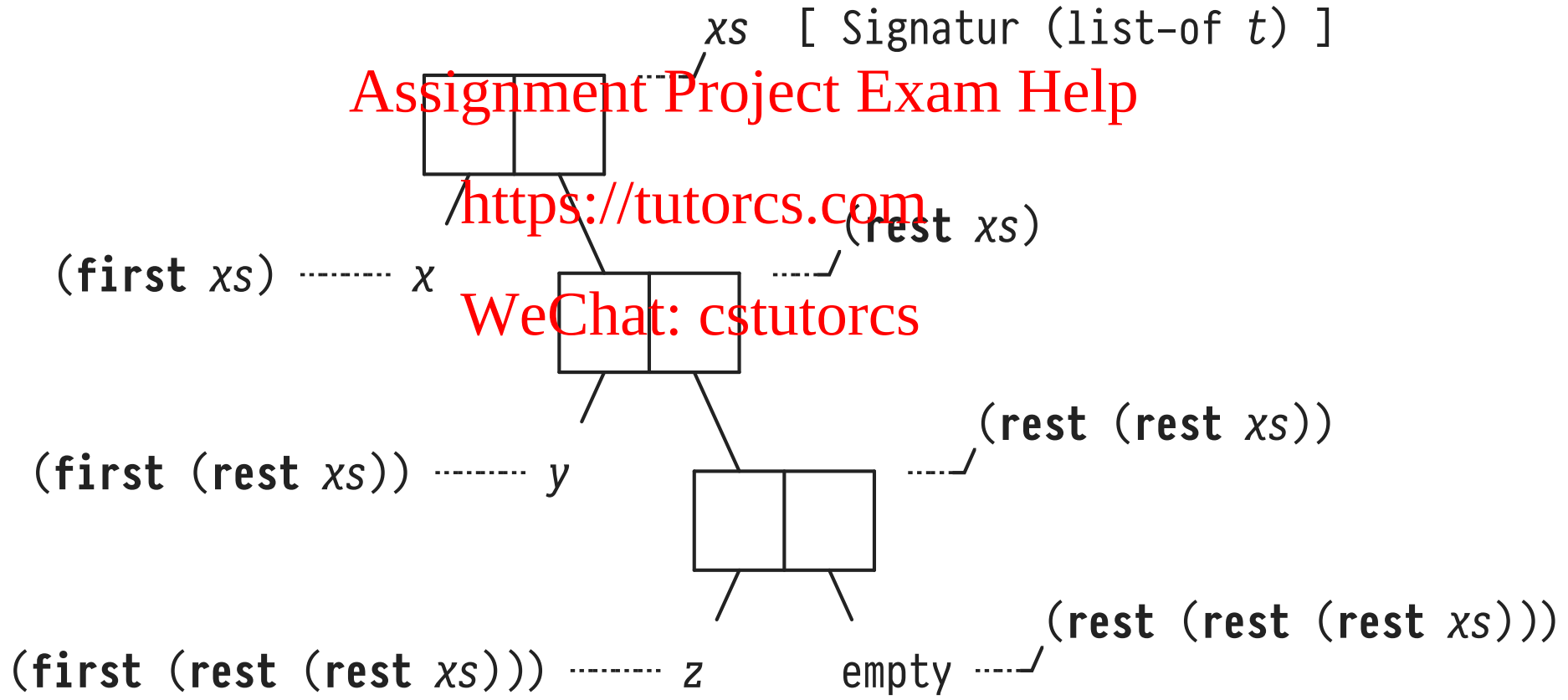
- Könnten *Nested Boxes* eine Darstellung von `(make-pair x (make-pair y (make-pair z empty)))` sein?

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- Diese Notation skaliert schlecht für längere Listen.

Visualisierung von Listen: Spines ("Rückgrat" der Liste)

Sei *xs* ≡ (make-pair *x* (make-pair *y* (make-pair *z* empty))).
Die **Spine** (auch: Rückgrat) der Liste *xs* ist:



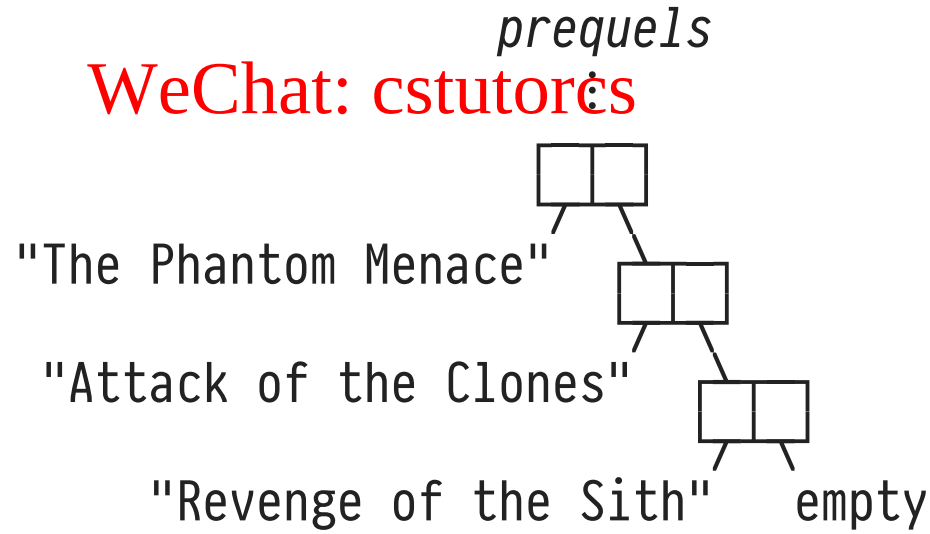
Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

Visualisierung von Listen: Spine-Beispiel

```
(: prequels (list-of string))
(define prequels
  (make-pair "The Phantom Menace"
     (make-pair "Attack of the Clones"
       (make-pair "Revenge of the Sith" empty))))
```

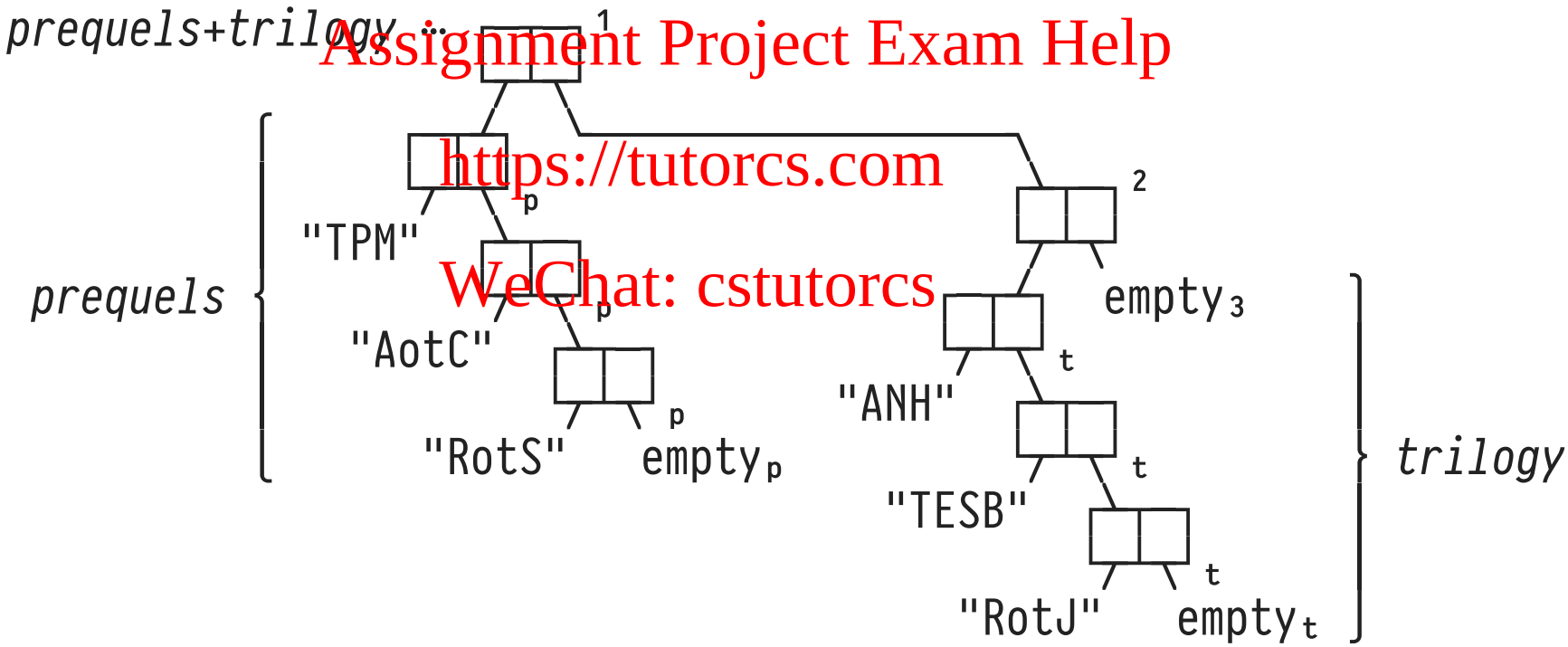
Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

- **Spine** der Liste *prequels*:



Visualisierung von Listen: Liste von Listen

```
(: prequels+trilogy (list-of (list-of string)))
(define prequels+trilogy
  (make-pair₁ prequels
     (make-pair₂ original-trilogy empty₃)))
```



Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

□p: aus *prequels*, □t: aus *trilogy*