

# PROGRAMMIERUNG

## ÜBUNG 13: $H_0$ – EIN EINFACHER KERN VON HASKELL

---

Eric Kunze

`eric.kunze@mailbox.tu-dresden.de`

1. Funktionale Programmierung
  - 1.1 Einführung in Haskell: Listen
  - 1.2 Algebraische Datentypen
  - 1.3 Funktionen höherer Ordnung
  - 1.4 Typpolymorphie & Unifikation
  - 1.5 Beweis von Programmeigenschaften
  - 1.6  $\lambda$ -Kalkül
2. Logikprogrammierung
3. Implementierung einer imperativen Programmiersprache
  - 3.1 Implementierung von  $C_0$
  - 3.2 Implementierung von  $C_1$
4. Verifikation von Programmeigenschaften
5.  **$H_0$  – ein einfacher Kern von Haskell**

## **$H_0$ – ein einfacher Kern von Haskell**

---

- ▶ **Ziel:** verstehe den Zusammenhang  $H_0 \leftrightarrow AM_0 \leftrightarrow C_0$
- ▶  $H_0$ : *tail recursive* Funktionen — rechte Seite enthält
  - ▶ keinen Funktionsaufruf
  - ▶ einen Funktionsaufruf an der äußersten Stelle (nicht verschachtelt)
  - ▶ eine Fallunterscheidung, deren Zweige wie oben aufgebaut sind

$H_0$  ist klein genug, dass es auf der  $AM_0$  laufen kann:

- ▶ Befehle bleiben die gleichen
- ▶ baumstrukturierte Adressen beginnen mit Funktionsbezeichner (z.B.  $f.1.3$ )
- ▶ Übersetzung von rechten Seiten  $\dots = exp$ :
  - ▶ Übersetze  $exp$
  - ▶ STORE 1 (ja – immer die 1)
  - ▶ WRITE 1
  - ▶ JMP 0
- ▶ Übersetzung von Funktionsaufrufen  $\dots = f\ x1\ x2\ x3$ :
  - ▶ LOAD  $x1$ ; LOAD  $x2$ ; LOAD  $x3$
  - ▶ STORE  $x3$ ; STORE  $x2$ ; STORE  $x1$  (umgekehrte Reihenfolge!)
  - ▶ JMP  $f$

$H_0$  (funktional) und  $C_0$  (imperativ) sind gleich stark – wir können Programme jeweils ineinander äquivalent übersetzen!

Standardisierung:

- ▶ keine Konstanten
- ▶ Es gibt  $m$  Variablen  $x_1, \dots, x_m$  ( $m \geq 1$ )
- ▶ Wir lesen  $k$  Variablen  $x_1, \dots, x_k$  ein ( $0 \leq k \leq m$ )
- ▶ Es gibt genau eine Schreibanweisung direkt vor `return`

- ▶ jedes Statement (in  $C_0$ ) erhält einen *Ablaufpunkt*
- ▶ jeder Ablaufpunkt  $i$  wird durch eine Funktion  $f_i$  (in  $H_0$ ) repräsentiert, die *alle* Programmvariablen als Argumente hat
- ▶ Funktionswerte beschreiben Veränderungen im Programmablauf

(einfaches) **Beispiel:**

- ▶ zwei Variablen  $x_1$  und  $x_2$
- ▶ betrachte Zuweisung  $x_2 = x_1 * x_1$  in  $C_0$
- ▶ Übersetzung zu  $f_1 \ x_1 \ x_2 = f_{11} \ x_1 \ (x_1 * x_1)$

Ein  $H_0$ -Programm kann in  $C_0$  mittels *einer* `while`-Schleife dargestellt werden. Dazu verwenden wir drei Hilfsvariablen:

- ▶ `flag` steuert den Ablauf der `while`-Schleife, d.h. wenn das  $H_0$ -Programm terminiert, wird `flag` falsch
- ▶ `function` steuert in einer geschachtelten `if-then-else`-Anweisung, welche Funktion ausgeführt wird
- ▶ `result` speichert den Rückgabewert der Funktion



# Übungsblatt 13

## *Aufgabe 1*

---

## AUFGABE 1 – TEIL (A)

$$f: \mathbb{N} \rightarrow \mathbb{N} \quad \text{mit} \quad f(n) = \sum_{i=1}^n \prod_{j=1}^i j$$

# AUFGABE 1 – TEIL (A)

$$f: \mathbb{N} \rightarrow \mathbb{N} \quad \text{mit} \quad f(n) = \sum_{i=1}^n \prod_{j=1}^i j$$

```
1 module Main where
2
3 --      i      sum      j      prod
4 f :: Int -> Int -> Int -> Int -> Int
5 f x1 x2 x3 x4
6   = if x3 > 1
7     then f x1 x2 (x3 - 1) (x3 * x4)
8     else if x1 > 0
9           then f (x1 - 1) (x2 + x4) (x1 - 1) 1
10          else x2
11
12 main = do x1 <- readLn
13           print (f x1 0 x1 1)
```

**Gegeben:**

```
1 f :: Int -> Int
2 f x1 = if x1 < 42
3       then x1
4       else if x1 > 42
5             then f (x1 `div` 2)
6             else 42
```

**Gesucht:** äquivalentes  $AM_0$ -Programm

## Gegeben:

```

1 f :: Int -> Int
2 f x1 = if x1 < 42
3       then x1
4       else if x1 > 42
5             then f (x1 `div` 2)
6             else 42
    
```

## Gesucht: äquivalentes $AM_0$ -Programm

```

f:      LOAD 1; LIT 42; LT; JMC f.3;
        LOAD 1; STORE 1; WRITE 1; JMP 0;
f.3:    LOAD 1; LIT 42; GT; JMC f.2.3;
        LOAD 1; LIT 2; DIV; STORE 1; JMP f;
f.2.3:  LIT 42; STORE 1; WRITE 1; JMP 0;
    
```

## Gegeben:

```

1 f1  x1 = if ((x1 'mod' 2) == 0) then f11 x1
2           else f12 x1
3 f11 x1 = f2 (x1 'div' 2)
4 f12 x1 = f2 (x1 - 1)
5 f2  x1 = f3 (2 * x1)
    
```

## Gesucht: äquivalentes $C_0$ -Programm

## Gegeben:

```

1 f1  x1 = if ((x1 'mod' 2) == 0) then f11 x1
2           else f12 x1
3 f11 x1 = f2 (x1 'div' 2)
4 f12 x1 = f2 (x1 - 1)
5 f2  x1 = f3 (2 * x1)
    
```

## Gesucht: äquivalentes $C_0$ -Programm

```

1 if ((x1 % 2) == 0)
2   x1 = x1 / 2;
3 else
4   x1 = x1 - 1;
5 x1 = 2 * x1;
    
```

# Übungsblatt 13

## *Aufgabe 2*

---



**Gegeben:**

```
1 h :: Int -> Int -> Int -> Int
2 h x1 x2 x3 = if x3 > x1
3               then (x2 - 1)
4               else h x2 (x1 - x3) x2
```

**Gesucht:** äquivalentes  $AM_0$ -Programm

## Gegeben:

```

1 h :: Int -> Int -> Int -> Int
2 h x1 x2 x3 = if x3 > x1
3               then (x2 - 1)
4               else h x2 (x1 - x3) x2

```

## Gesucht: äquivalentes $AM_0$ -Programm

```

1 h:    LOAD 3; LOAD 1; GT; JMC h.3;
2        LOAD 2; LIT 1; SUB; STORE 1; WRITE 1; JMP 0;
3 h.3:  LOAD 2; LOAD 1; LOAD 3; SUB; LOAD 2;
4        STORE 3; STORE 2; STORE 1; JMP h;

```

## Lösung:

```
A:  scanf("%d", &x1);
    x1 = 3 + x1;
    x2 = 5;
    flag = 1;
```

```
B:  x2 == x1
```

```
C:  result = 30;
    flag = 0;
```

```
D:  result = x2;
    flag = 0;
```

```
E:  function == 2
```

```
F:  if (10 <= x2) {
        x1 = x1 - x2;
        x2 = x2 - 1;
    } else {
        x1 = x1 + x2;
        x2 = 10;
        function = 1;
    }
```