Tail Recursion

Tail Rekursion
Motivation

- Auswertung von Rekursion auf Stack am Beispiel:

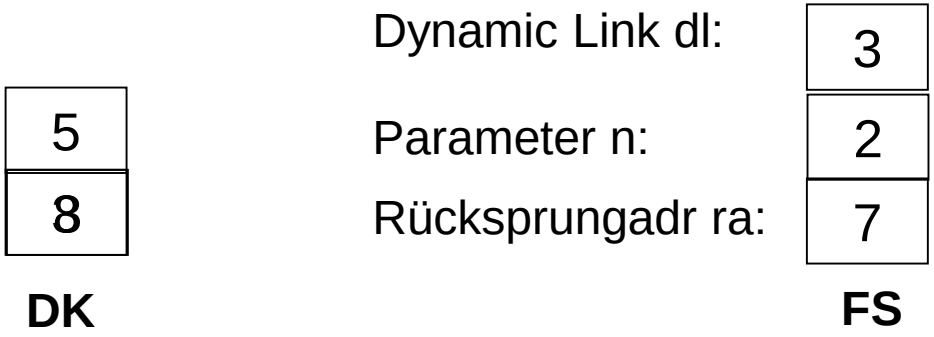
```
def fak(n: Int) : Int = if (n==0) 1 else n*fak(n-1)
```

- Auswertung von Ausdrücken in Postfixnotation auf Datenkeller DK:

```
3 + 5  ->   3 5 +  ->   LIT 3; LIT 5; ADD
```

- Pro Funktionsaufruf ein Block auf Funktionsstack FS

```
fak(2) ->
```



Tail Rekursion
Auswertung von fak(2)

```
def fak(n: Int) : Int = if (n==0) 1 else n*fak(n-1)
```

Initialblock auf FS

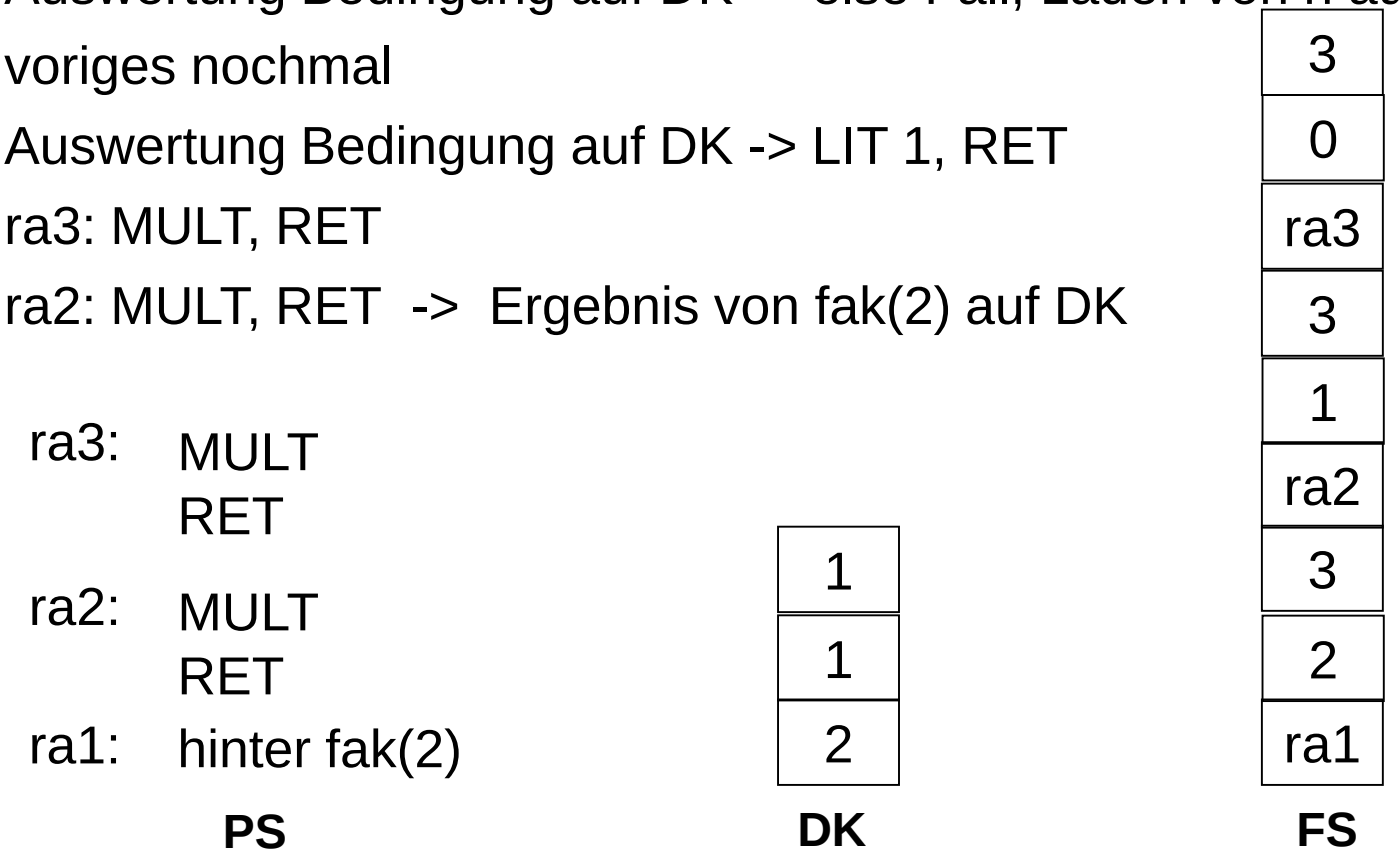
Auswertung Bedingung auf DK -> else Fall, Laden von n auf DK, fak(n-1)

voriges nochmal

Auswertung Bedingung auf DK -> LIT 1, RET

ra3: MULT, RET

ra2: MULT, RET -> Ergebnis von fak(2) auf DK



Tail Rekursion
Aufwand und Problemstellung

- Aufwand:
  - Zeit: linear
  - Speicher: linear
- Problem:
  - letzter Befehl bei Funktionsaufruf: MULT
  - Multiplikationen erst ganz am Ende ausgeführt
  - Funktionsaufruf zwischendurch
  - voriger Funktionsblock noch nicht gelöscht, wenn neuer aufgelegt

Tail Rekursion
Idee und Lösung

- Idee:
  - lege erst neuen Funktionsblock auf, wenn voriger gelöscht (oder verwende gleichen)
  - immer nur ein Funktionsblock auf FS
  - Speicherbedarf konstant
- Klappt nur, wenn
  - **nur 1 Funktionsaufruf auf rechter Seite,**
  - **der ganz am Ende ausgeführt wird (bei Postfix-Auswertung)**

Tail-Rekursion
Definition

Definition (Tail-Rekursiv = endrekursiv) aus Wikipedia:

Eine rekursive Funktion *f* ist **endrekursiv** (englisch *tail recursive*; auch **endständig rekursiv**, **iterativ rekursiv**, **repetitiv rekursiv**), wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von *f* ist.

Vorteil dieser Funktionsdefinition ist, dass kein zusätzlicher Speicherplatz zur Verwaltung der Rekursion benötigt wird.

Iteration -> Tail Rekursion
Motivation

- Bei fak
  - Zeit : linear -> linear
  - Platz: linear -> konstant
- Schlimmer noch, wenn sowohl Zeit-, wie Speicherbedarf exponentiell
- Fibonacci rekursiv:

```
def fib(n: Int) : Int = if (n==0 || n==1) n
                        else fib(n-1) + fib(n-2)

fib(4)  ->   Int = 5
```

- iterative Lösung: viel besser

- Lösung: iterativ -> Tail-rekursiv

Iteration -> Tail Rekursion
Ausgangspunkt

- Fibonacci iterativ:

```
public int fibIterative(int n) {
    int vorvor = 0, vor = 1, help;

    for (int i = 1; i <= n; i++) {
        help = vorvor + vor;
        vorvor = vor;
        vor = help;
    }
    return vorvor;
}
```

- Zeit linear
- Platz konstant

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

Iteration -> Tail Rekursion
Umsetzung

- gehe von iterativem Programm aus
- lokalen Variablen -> Parameter
- Variablenzuweisungen -> in Parametern ausführen

```
def fibTR(n: Int) : Int = fibHelp(n,0,1)
def fibHelp(n: Int, vorvor: Int, vor: Int) : Int
   =  if (n==0) vorvor
      else fibHelp(n-1,vor,vorvor+vor)
```

- kann sich `help` sparen, wegen call-by-value

- Zeit linear
- Platz konstant
  -> genauso gut, wie iterative Lösung