3AA

Prozeduren und Rekursion

29.11.05

Prof. Dr. Wolfgang P. Kowalk
Universität Oldenburg

WS 2005/2006

3AA Prozeduren

- Berechnete Sprungadresse
  - Ausführung bestimmter Anweisungen durch Schleifen
  - Stattdessen: Rufe Anweisungen 'aus Entfernung' auf
    - Wo stehen die Anweisungen? Zieladresse!
    - Wohin, wenn man 'fertig' ist? Rücksprungadresse!
    - Beide Adressen können variabel sein!

3AA Prozeduren

```
 1            // Computed Goto
 2            nop
 3            adr Ziel := val Wert - con 'A'
 4            adr Ziel := val Ziel + con Start
 5            goto val Ziel
 6 Start      goto con MarkeA
 7            goto con MarkeB
 8            goto con MarkeC
 9            goto con MarkeD
10            goto con MarkeE
11 MarkeA     adr Ergebnis := con 1
12            stop
13 MarkeB     adr Ergebnis := con 2
14            stop
```

3AA Prozeduren

```
11 MarkeA     adr Ergebnis := con 1
12            stop
13 MarkeB     adr Ergebnis := con 2
14            stop
15 MarkeC     adr Ergebnis := con 3
16            stop
17 MarkeD     adr Ergebnis := con 4
18            stop
19 MarkeE     adr Ergebnis := con 5
20            stop
21
22 Ziel
23 Wert       con 'C'
24 Ergebnis
```

3AA Prozeduren

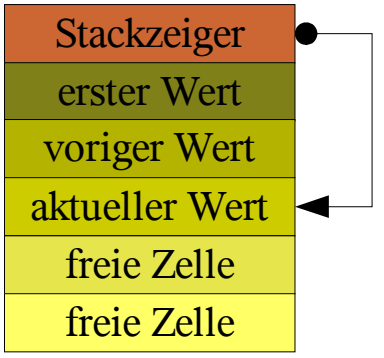
```
 1           // Rücksprungadresse bei Subroutinen
 2           adr Parameter :=  con 5
 3   adr Ruecksprungadresse := con PROGRAMCOUNT + con 2 // setze die Rücksprungadresse
 4           goto con Quadriere  // Springe in die Subroutine
 5           adr Ergebnis := val RueckgabeWert  // an diese Stelle wird zurückgesprungen
 6           step
 7
 8           adr Parameter :=  con 9
 9   adr Ruecksprungadresse := con PROGRAMCOUNT + con 2 // setze die Rücksprungadresse
10           goto con Quadriere  // Springe in die Subroutine
11           adr Ergebnis := val RueckgabeWert  // an diese Stelle wird zurückgesprungen
12           stop
13
14 Parameter
15 RueckgabeWert
16 Ruecksprungadresse
17 Ergebnis
18
19 Quadriere adr RueckgabeWert := val Parameter * val Parameter
20           goto val Ruecksprungadresse  // springe an die gespeicherte Rücksprungadresse
```

3AA Prozeduren

- Prozeduren
  - Ausführung bestimmter Anweisungen durch Schleifen
  - Stattdessen: Rufe Anweisungen 'aus Entfernung' auf
- Vorteile von Prozeduren
  - Anweisungsfolge nur einmal schreiben
    - spart Speicher, aber Inline-Methoden? (veraltet)
    - strukturiert Programm! Sehr wichtig!
    - dokumentiert Funktionalität bei geeignetem Namen! Sehr wichtig!
    - Programm kann einfacher geändert werden!
      - korrigiert
      - ersetzt usw.

3AA Prozeduren

- Prozeduren
  - In Fortran: Subroutine
    - Prozedur nur einmal aufrufbar (keine Rekursion)
    - Werteübergabe nur über Parameterliste
  - In Algol 60 (und meisten späteren Sprachen)
    - Prozedur kann sich (in-) direkt selbst aufrufen
    - Rücksprungadresse?
    - Parameter?
  - Lösung: Stack (engl. Stapel)



3AA Prozeduren

```
1             // Fakultät rekursiv mit STACK
   2
   3              nop
   4              push con 4 // Schiebe Parameter auf STACK
   5              jumpProc con Fak // rufe Prozedur auf
   6 Adresse1     getPull adr ergebnis
                                  // hole Ergebnis vom STACK
   7              pull con 1       // setze STACK zurück
   8              stop
   9
  10 Fak          get con 1, adr par // lade Parameter vom STACK
  11              if val par <= con 1 then goto con Fertig1
                                     // ? zu Ende ?
  12              push val par - con 1 // Schieb Datum auf STACK
  13              jumpProc con Fak     // Fak(par)
  14 Adresse2     get con 0, adr par2  // := Fak(par)
```

3AA Prozeduren

```
10 Fak        get con 1, adr par // lade Parameter vom STACK
11            if val par <= con 1 then goto con Fertig1
                                    // ? zu Ende ?
12            push val par - con 1 // Schieb Datum auf STACK
13            jumpProc con Fak     // Fak(par)
14 Adresse2   get con 0, adr par2  // := Fak(par)
15            pull con 1           // setze STACK zurück
16            get con 1, adr par   // hole Parameter
17            put con 1, val par * val par2
                                 // Schiebe N * (N-1)! auf STACK
18            returnProc                 // Fertig
19
20 Fertig1    put con 1, con 1  // Schiebe 1 auf STACK
21 Fertig     returnProc        // Fertig, spring an Stack-
Adresse
```

Rekursive Funktionen

- In Java
  - ```
    int faculty(int parameter) {
      if(parameter<=1) return 1;
      return parameter*faculty(parameter-1);
    }
    ```
  - $F_k = k \cdot F(k\text{-}1)$, $F_0 = F_1 = 1$
  - Löse Problem für einfache, explizite Aufgabe
    - z.B. **`parameter<=1`**
  - Formuliere Lösung für 'etwas' einfacheres Problem
    - z.B. **`parameter*faculty(parameter-1)`**
    ```

3AA Prozeduren

```
 1      // Fakultät rekursiv mit STACK
 2      push val ergebnis          // Schiebe Parameter auf STACK
 3      jumpProc con Fak           // rufe Prozedur auf
 4      getPull adr ergebnis       // hole Ergebnis vom STACK
 5      stop
 7 Fak  get con 1, adr par         // lade Parameter vom STACK
 8      if val par <= con 1 then goto con Fertig1 // ? zu Ende
 9      push val par - con 1       // Schiebe Parameter auf STACK
10      jumpProc con Fak           // Fak(par)
11      getPull adr par2           // := Fak(par)
12      get con 1, adr par         // hole Parameter
13      put con 1, val par * val par2 // N * (N-1)! auf STACK
14      returnProc                 // Fertig
15
16 Fertig1   put con 1, con 1          // Schiebe 1 auf STACK
17 Fertig    returnProc                // Fertig
```

Rekursive Funktionen

- Neue Anweisungen in 3AA
  - **`push con 1`**
    `push val wert`
    `push val val zeiger`
    - Erhöhe **`STACK`** um 1, schreibe Wert in Stack
  - **`pull con 1`**
    - Erniedrige **`STACK`** um 1
  - **`getPull adr wert`**
    `getPull adr adr zeiger`
    - Schreibe Wert nach **`wert`**, erniedrige **`STACK`** um 1

Rekursive Funktionen

- Neue Anweisungen in 3AA
  - `put con 1, con 0 // Mem[STACK-1] ← 0`
    `put con 0, val wert // Mem[STACK+0] ← wert`
    `put con 2, val val zeiger + val wert`
    - Schreibe an Adresse `[STACK]-k` den Wert
    - Der `STACK` wird nicht verändert!
  - `get con 1, adr wert // Mem[STACK-1] → wert`
    `get con 0, adr wert2 // Mem[STACK+0] → wert2`
    `get con 2, adr adr zeiger // Mem[STACK-2] → [zeiger]`
    - Schreibe den Wert an Adresse `[STACK]-k`
    - Der `STACK` wird nicht verändert!

Rekursive Funktionen

- Idee zur Rekursion
  - Löse Problem für expliziten Fall (0, 1)
- Löse Problem für komplexes Problem (n):
  - Formuliere Lösung für Problem (n-1)
  - Berechne Lösung für (n) aus Lösung für (n-1)
  - Allgemeines Lösungskonzept
    - Probleme
      - Laufzeit
      - Speicher für Stack

Rekursive Funktionen

- Rekursion
  - Vergleich mit 'Mathematischer Induktion'
- Beweise Behauptung für Aussage $A(n)$:
  - Zeige Aussage für $n=1$: $A(1)$
  - Nehme an, Aussage gelt für $n=k$: $A(k)$
    - Zeige dann: Aussage gilt für $n=k+1$: $A(k+1)$
  - Logisches Beweiskonzept aus der (Meta-)Mathematik
    - $A(1) \Rightarrow A(2) \Rightarrow A(3) \Rightarrow \ldots \Rightarrow A(k\text{-}1) \Rightarrow A(k)$
      k endlich!

Rekursive Funktionen

- Probleme bei Rekursion
  - Effizienz
    - Heute meist kein Problem mehr!
  - Laufzeitkomplexität
    - Beispiel: Fibonacci-Zahlen
      - Folgen von Zahlenwerte, mit

        $f_n = f_{n-1} + f_{n-2}, f_0 = f_1 = 1.$

        1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..

Rekursive Funktionen

- Beispiel: Fibonacci-Zahlen
  - Iterative Implementierung
    - ```
      int fib = 1; fib0 = 0;
      while ... {
        fib  = fib + fib0;
        fib0 = fib - fib0;
      }
      ```
  - Rekursive Implementierung
    - ```
      int fibonacci(int n) {
        if(n<=1) return 1;
        return fibonacci(n-1)+fibonacci(n-2);
      }
      ```
  - Mathematische Wertebeschreibung

$$f_k = \frac{1+\sqrt{5}}{2\cdot\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^k + \frac{\sqrt{5}-1}{2\cdot\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^k$$

Rekursive Funktionen

- Probleme bei Rekursion
  - Beispiel: Fibonacci-Zahlen
    - Laufzeit exponentiell! Beweis
      - $H(n)$: Laufzeit für Parameter n.
      - $H(0)=H(1)=1$ (ein Funktionsaufruf)
      - Rekursive Definition: $H(n) = 1 + H(n\text{-}1) + H(n\text{-}2)$

$H(k) = 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177$

$F(k) = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$

$H(n) = 2 \cdot F(n) - 1$!

$$\begin{aligned} H(k+1) &= 1 + H(k) + H(k\text{-}1) \\ &= 1 + 2 \cdot F(k) - 1 + 2 \cdot F(k\text{-}1) - 1 \\ &= 2 \cdot F(k) + 2 \cdot F(k\text{-}1) - 1 \\ &= 2 \cdot F(k+1) - 1 \end{aligned}$$

$$H(k) \approx 2{,}34 \cdot 1{,}62^{k} - 0{,}341 \cdot (-0{,}62)^{k} - 1$$

Rekursive Funktionen

- Probleme bei Rekursion
  - Beispiel: Fibonacci-Zahlen
    - Laufzeit exponentiell!
      - $H(n)$: Laufzeit für Parameter n.
      - $H(0)=H(1)=1$ (ein Funktionsaufruf)
      - Rekursive Definition: $H(n) = 1 + H(n\text{-}1) + H(n\text{-}2)$
      - $H(k) \approx 2{,}34 \cdot 1{,}62^{k} - 0{,}341 \cdot (-0{,}62)^{k} - 1$
      - ```
        int H = 1, M = 1;
        do {
         H = H + M + 1
         M = H - M - 1
        } while (...)
        ```
        ```

Rekursive Funktionen

- $N1=1.618033988749895$, $N2=-0.618033988749894 9$
- $U_a=2.341640786499873 6$, $V_a=-0.341640786499873 8$, $W_a=-1$
- $U_b=1.447213595499957 7$, $V_b=0.552786404500042$, $W_b=-1$
- $a_k = U_a*N_1^k+V_a*N_2^k+W_a$
- $b_k = U_b*N_1^k+V_b*N_2^k+W_b$
  - $a_0=1.0$, $b=0.999999999999998$
  - $a_1 = 3.0$, $b=1.0$
  - $a_2 = 5.0$, $b=2.999999999999999$
  - $a_3 = 8.999999999999998$, $b=4.999999999999999$
  - $a_4 = 15.0$, $b=8.999999999999998$
  - $a_5 = 25.0$, $b=14.999999999999996$
  - $a_6 = 40.999999999999999$, $b=24.999999999999996$
  - $a_7 = 67.0$, $b=40.999999999999999$
  - $a_8 = 109.000000000000001$, $b=67.0$
  - $a_9 = 177.0$, $b=109.0$