Programmiersprache C

Grundlegende Konstrukte

Variablen

Haskell

- Benannter Ausdruck eines Zwischenergebnisses
- Unveränderbar nach Vollzug der Bindung

C

- Benannter Speicherplatz für Zwischenergebnisse
- Allokation und Initialisierung getrennt
- Überschreiben per Zuweisung möglich

Beispiele von Variablen

```
z :: Int
z = 2
z = 3 -- geht nicht
```

```
int z; /* Deklaration */
z = 3; /* Zuweisung */
z = z - 1; /* z -= 1; z-- */
```

```
f :: Int -> Int -> Int
f = \x -> \y -> 2*x + y
f x = \y -> 2*x + y
f x y = 2*x + y
```

```
int f(int x, int y) {
    return 2*x + y;
}
```

Funktion != Variable

IKS

Funktionen

- Alle Arbeit wird in Funktionen erledigt
- Hauptfunktion ist immer nur ein Seiteneffekt, keine Berechnung: Die Programmfunktionalität

Haskell

- pur: Gleiche Parameter = Gleiches Ergebnis
- Variablen höheren Typs

C

- Seiteneffekte: Verschiedene Ergebnisse möglich

Funktionsbeispiele

```
doppel :: Int -> Int
doppel x = x + x
```

```
int doppel(int x) {
    return x + x;
}
```

```
f :: Int -> Int -> Int
f x y = doppel x + y
```

```
int f(int x, int y) {
    return doppel(x) + y;
}
```

IKS

Funktionen mit Variablen

```
f :: Int -> Int -> Int
f x y = res'
    where
        res  = doppel x
        res' = res + y
```

```
int f(int x, int y) {
    int res = doppel(x);
    res += y;
    return res;
}
```

Immer neue Variablen, da nur Ausdrücke

Variable = Speicherplatz wird verändert

Rekursion und Schleifen

- Rekursion: siehe Endrekursion, siehe Rekursion
- Endrekursion: siehe Endrekursion
- Rekursion ist der Aufruf einer Funktion aus sich selbst heraus => Ausbildung von Schleifen
- Endrekursion ist der Aufruf der ursprünglichen Funktion mit unverändertem Durchreichen des Ergebnisses => Sprung an den Anfang
- C: End-/Rekursion ist immer/besser zu entfernen

Endrekursion

```
ggt :: Int -> Int -> Int
ggt x y
    | x  <  y      = ggt y x
    | y == 0       = x
    | otherwise =
          ggt y (x 'mod' y)
```

```
int ggt(int x, int y) {
    if(x < y) {
        return ggt(y,x);
    } else if(y == 0) {
        return x;
    } else {
        return ggt(y, x%y);
    }
}
```

IKS

Endrekursion => while-Schleife

```
int ggt(int x, int y) {
    if(x < y) {
        int t; t=x; x=y; y=t;
    }
    if(y == 0) {
        return x;
    } else {
        return ggt(y, x%y);
    }
}
```

```
int ggt(int x, int y) {
    if(x < y) {
        int t=x; x=y; y=t;
    }
    while(y != 0) {
        int t=x; x=y; y=t%y;
    }
    return x;
}
```

IKS

Rekursion

```
fac :: Int -> Int
fac 0 = 1
fac x = x * fac (x-1)
```

```
int fac(int x) {
    if(x == 0) {
        return 1;
    } else {
        return x * fac(x-1);
    }
}
```

Wie wird berechnet?

Häufiges Konstrukt – for-Schleife

```
int fac(int x) {
    int res = 1;
    while(x > 0) {
        res *= x;
        x--; /* x = x – 1 */
    }
    return res;
}
```

```
int fac(int x) {
    int res;
    for(res = 1; x > 0; x--) {
        res *= x;
    }
    return res;
}
```

Mehrfache Rekursion

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

Lange Rechenzeit, da mehrfache Aufrufe

```
fib :: Int -> Int
fib x = fibs !! x

fibs = 1 : 1 :
    [ fib (x-1) + fib (x-2)
    | x <- [2..] ]
```

Array zum Speichern aller Zwischen-ergebnisse

IKS

Array für Zwischenresultate

```
int fib(int x) {
    int fs[50]; /* 0 .. 49 */
    int i;

    if(x < 0 || x > 49) {
        return -1; /* zu klein */
    }

    /* Initialisierung */
    fs[0] = fs[1] = 1;
    for(i=2; i<=x; i++) {
        fs[i] = fs[i-1] + fs[i-2];
    }
    /* Ergebnis ausgeben */
    return fs[x];
}
```

IKS

Funktionen und Blöcke in C

```
/* Kommentar */
typ funktion(typ name, typ name, ...) block
Das Folgende ist ein Block:
{
    deklarationen;
    befehle;
}
```

Datenflußsteuerung in C

while(test) block

do block while(test)

if(test) block else block

```
switch(ausdruck) {
    case wert: befehle;
               break;
    default: befehle;
}
```

for(befehl; test; befehl)

    block

label: befehle;

goto label;

continue – next loop

break – exit loop

test ? ausdruck : ausdruck

if-then-else als Ausdruck

IKS

Operationen in C

- Numerisch: + - * /(durch) %(modulo)
- Vergleiche: < <= == != >= >
- Bitweise: &(und) |(oder) ^(xor) ~(invert)
- Logisch (0 == falsch): &&(und) ||(oder)
- Strukturen: .(Elementzugriff)
- Zeiger: &(Adresse) *(Wert) ->(Strukturelement)
- Typkonvertierung: (typ)(Ausdruck)

Datenstrukturen

- Zusammenfassung einfacher Elemente zu komplexeren Typen
- Ziel: Erhaltung von Zusammenhängen
- Grundbausteine: Struktur und Union
- Struktur: Menge mehrerer Elemente
- Union: Auswahl eines von mehreren Elementen
- Enum: Aufzählung ohne interne Daten
  => Spezialfall von Union, real eine Zahl

Datenstrukturen

```
data Point = Punkt {
    x, y :: Int
    c :: Color
}
```

```
struct Point {
    int x, y;
    enum Color c;
};
```

```
data Color
    = Red | Green | Blue
```

```
enum Color {
    Red, Green, Blue
};
```

IKS

Datenstrukturen

```
data Ausdruck
    = Var {
        c :: Char
        i :: Int
    }
    | Val { i :: Int }
```

```
struct Ausdruck {
    enum {Var, Val} typ;
    union {
        struct {
            char c;  /* u.var.c */
            int i;   /* u.var.i */
        } var;
        int val;     /* u.val */
    } u;
};
```

IKS

Morphismen
Arbeiten mit Datenstrukturen

- Morphismus: An der Gestalt/Form orientiert
- Anamorphismus: Erzeugung
  - Griechisch: ανα = aufwärts (vgl. Anabolika)
- Katamorphismus: Zerstörung
  - Griechisch: κατα = abwärts (vgl. Katastrophe)
- Hylomorphismus: Strukturen als Zwischenschritt
  - Griechisch: υλοσ = Staub/Materie
  - Erzeugen und zerstören: hylo f g = kata f . ana g

IKS

Felder – Arrays

- Indizierte Menge von Werten gleichen Typs
  - Haskell: feld :: [a] polymorph, unbegrenzt
  - C: int feld[100]; feste Größe, fester Typ
- Schneller Zugriff auf Werte zum Lesen und Überschreiben => O(1) für alle Operationen
- In C Übergabe an Funktionen mittels Zeiger und Länge: void sort(int * daten, int anzahl);
- In C sind Felder und Zeigerarithmetik gleich

Katamorphismus – Fold

```
foldl f y [] = y
foldl f y (x:xs) =
    foldl f (f y x) xs
```

```
r = y;
for(i = 0; i < len; i++) {
    r = f(r, feld[i]);
}
```

```
foldr f y [] = y
foldr f y (x:xs) =
    f x (foldr f y xs)
```

```
r = y;
for(i = len; i-- > 0;) {
    r = f(feld[i], r);
}
```

IKS

Rekursive Datenstrukturen

- Rekursive Datenstrukturen enthalten sich selbst als Elemente
- Direkter Speicherbedarf wäre unendlich
- Haskell: Speicher erst bei Bedarf angefordert
- C: Zeiger auf die Datenwiederholung
- Zeiger können leer sein => null (Haskell: Maybe)
- Zeigeralgebra: plus/minus (in Datenstrukturen)
- Speicher händisch verwalten (malloc, free)

Rekursive Datenstrukturen

```
data Baum
    = Blatt { x :: Int }
    | Knoten {
        x :: Int
        l, r :: Baum
    }
```

```
struct Baum {
    enum {Blatt, Knoten} t;
    union {
        int blatt;
        struct {
            int x;
            struct Baum *l, *r;
        } knoten;
    } u;
};
```

IKS

Rekursive Datenstrukturen

```
data Baum = Baum {
    x :: Int
    l, r :: Maybe Baum
}
```

```
struct Baum {
    int x;
    struct Baum *l, *r;
};
```

- null zeigt an, daß kein Knoten vorliegt
- Viel einfachere Struktur aber Disziplin nötig:
  - l und r existieren nur zusammen oder gar nicht

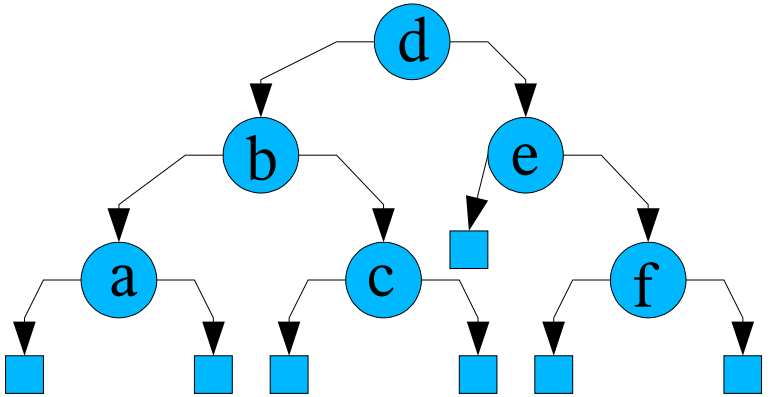
Katamorphismus – Traversierung

```
pre (Blatt i) = [i]
pre (Knoten i l r) =
    [i] ++ pre f l ++ pre f r
post (Blatt i) = [i]
post (Knoten i l r) =
    post f l ++ post f r ++ [i]
in (Blatt i) = [i]
in (Knoten i l r) =
    in f l ++ [i] ++ in f r
```

```
void pre(struct Baum * p) {
    printf ("%i ", p->x);
    if(p->l) { pre(p->l); }
    if(p->r) { pre(p->r); }
}
void in(struct Baum * p) {
    if(p->l) { in(p->l); }
    printf ("%i ", p->x);
    if(p->r) { in(p->r); }
}
```

IKS

Katamorphismus – Beispiel



- Beispiele für verschiedene Traversierungen:
  - Preorder “dbacef“
  - Postorder “acbfed“
  - Inorder “abcdef“
  - Levelorder “dbeacf“