1 Einleitung

Die Geschichte und Entwicklung von Programmiersprachen kann aufgefasst werden als Bestreben, immer stärker von der konkreten Hardware der Rechner zu abstrahieren. In der folgenden Tabelle sind einige Meilensteine dieser Entwicklung aufgeführt:

Sprachen/Konzepte:	Abstraktion von:
Maschinencode	
Assembler (Befehlsnamen, Marken)	Befehlscode, Adresswerte
Fortran: arithmetische Ausdrücke	Register, Auswertungsreihenfolge
Algol: Kontrollstrukturen, Rekursion	Befehlszähler, "goto"
Simula/Smalltalk/Java: Klassen, Objekte	Implementierung ("abstrakte Datentypen") Speicherverwaltung
Deklarative Sprachen: Spezifikation von Eigenschaften	Ausführungsreihenfolge $\hookrightarrow$ Optimierung $\hookrightarrow$ Parallelisierung $\hookrightarrow$ keine Seiteneffekte

Eine der wichtigsten charakteristischen Eigenschaften deklarativer Sprachen ist die **referenzielle Transparenz** (engl. **referential transparency**):

- Ausdrücke dienen *nur* zur Wertberechnung
- Der Wert eines Ausdrucks hängt nur von der Umgebung ab und nicht vom Zeitpunkt der Auswertung
- Ein Ausdruck kann durch einen anderen Ausdruck gleichen Wertes ersetzt werden (**Substitutionsprinzip**)

Eigentlich ist dies nichts besonderes, sondern ein aus der Mathematik bekanntes und übliches Prinzip:

> Der Wert von $x^2 - 2x + 1$ hängt nur vom Wert von x ab.

Hier sind Variablen also keine „Speicherzeillen", sondern:

- Variablen sind Platzhalter für Werte
- eine Variable bezeichnet immer den gleichen Wert

Die Programmiersprache C ist nicht referenziell transparent:

- Hier sind Variablen Namen von Speicherzellen
- Deren Werte (Inhalte) können verändert werden:

  ```
  x = 3;
  y = (++x) * (x--) + x;
  ```

  Hier bezeichnen das erste und das letzte Vorkommen der Variablen `x` *verschiedene* Werte!
  Außerdem ist der Wert von `y` nach der Ausführung abhängig von der Auswertungsreihenfolge!

Die referenzielle Transparenz ist verwandt mit dem **Substitutionsprinzip**:

- Es ist ein natürliches Konzept aus der Mathematik:
  Beispiel: Ersetze π immer durch 3,14159...
- Es ist fundamental für die Beweisführung („ersetze Gleiches durch Gleiches“, vereinfache Ausdrücke)
- Es vereinfacht die
  - Optimierung
  - Transformation
  - Verifikation

  von Programmen

Als Beispiel schauen wir uns einmal die *Tranformation und Optimierung* von Programmen an. Betrachten wir dazu folgende Funktionsdefinition in einer imperativen Sprache:

```
function f(n:nat) : nat =
begin
   write("Hallo");
   return (n * n)
end
```

Wenn wir die Anweisung

```
z := f(3)*f(3);
```

ausführen, erhalten wir als Ergebnis den Speicherwert `z=81` *und* die Ausgabe `HalloHallo`.

Nun könnte man auf die Idee kommen, den doppelten (identischen!) Funktionsaufruf wie folgt zu optimieren:

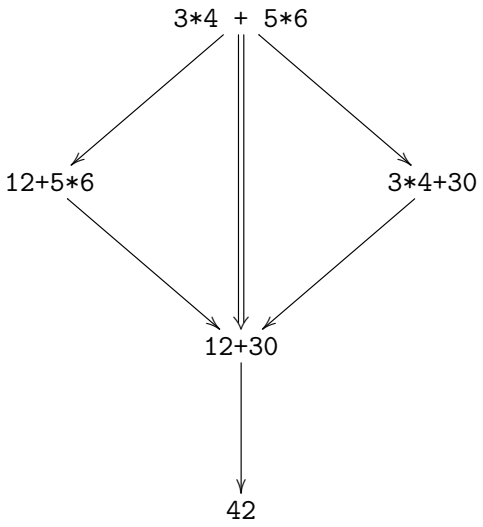
```
x:=f(3); z:=x*x;
```

Dies ist aber falsch, weil dann der Seiteneffekt nur einmal ausgeführt wird! Wir können also festhalten:

Seiteneffekte erschweren die Programmoptimierung!

Ein weiterer Vorteil der Seiteneffektfreiheit oder referenziellen Transparenz ist die Möglichkeit, *Programme flexibler auszuwerten*:

Betrachten wir hierzu verschiedene Auswertungsmöglichkeiten des Ausdrucks `3*4 + 5*6`:



Hierbei bezeichnet der Doppelpfeil in der Mitte die parallele Auswertung des Ausdrucks. Durch die flexible Auswertungsreihenfolge, die durch referenzielle Transparenz ermöglicht wird, kann man somit **parallele Rechnerarchitekturen** leichter ausnutzen.

Betrachten wir einen weiteren wichtigen Aspekt deklarativer Sprachen: Die Erstellung **lesbarer und zuverlässiger Programme**. Hierzu beginnen wir mit zwei Zitaten:

- Programme müssen so geschrieben werden, damit Menschen sie lesen und modifizieren, und nur nebenbei, damit Maschinen sie ausführen [Abelson/Sussman/Sussman 96]
- Programmierung ist eine der schwierigsten mathematischen Tätigkeiten [Dijkstra]

Die **Abstraktion von Ausführungsdetails verbessert die Lesbarkeit** von Programmen, wie wir an einigen kleinen Beispielen sehen.

Beispiel: Fakultätsfunktion:
Eine Formulierung in einer imperativen Programmiersprache könnte wie folgt aussehen:

```
function fac(n:nat):nat =
begin
   z:=1; p:=1;
   while z<n+1 do
   begin p:=p*z: z:=z+1 end;
   return(p)
end
```

Diese Implementierung birgt einige potenzielle Fehlerquellen:

- Initialisierung (`z:=1` oder `z:=0`?)
- Abbruchbedingung (`z<n+1` oder `z<n` oder `z<=n`?)
- Reihenfolge im Schleifenrumpf (zuerst oder zuletzt `z:=z+1`?)

Bei einer **deklarativen Formulierung** definieren wir dagegen nur die Eigenschaften von `fac`:

```
fac(0) = 1
fac(n) = n * fac(n-1)  if n>0
```

⇒ kein Zähler (`z`), kein Akkumulator (`p`), Rekursion statt Schleife, weniger Fehlerquellen

Beispiel: **Quicksort**

Imperativ (nach Wirth: Algorithmen und Datenstrukturen):

```
procedure qsort (l, r: index);
var i,j: index; x,w: item
begin
   i := l; j := r;
   x := a[(l+r) div 2]
   repeat
      while a[i]<x do i := i+1;
      while a[j]>x do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1;
      end
   until i > j;
   if l < j then qsort(l,j);
   if j > r then qsort(i,r);
end
```

Hier gibt es wieder viele potenzielle Fehlerquellen, wie Abbruchbedingungen, Reihenfolgen,...

Dagegen sieht eine *deklarative Formulierung* (in diesem Fall funktional in der Programmiersprache Haskell) wie folgt aus (hierbei werden Listen und nicht Felder sortiert, wobei `[]` die leere Liste bezeichnet und `(x:xs)` eine Liste mit dem erstem Element `x` und der Restliste `xs`):

```
qsort [] = []
qsort (x : xs) = qsort (filter (<x) xs)
              ++ [x]
              ++ qsort (filter (>=x) xs)
```

Wie man sieht, findet sich hier die Idee von Quicksort (Teilen und Zusammensetzen) klar im Programmcode wieder.

Datenstrukturen und Speicherverwaltung

Viele Programme benötigen komplexe Datenstrukturen
Imperative Sprachen: Zeiger → fehleranfällig ("core dumped")

Beispiel: Datenstruktur **Liste**:
- entweder leer (NIL)
- oder zusammengesetzt aus 1. Element (Kopf) und Restliste

Aufgabe: Gegeben zwei Listen $[x_1, \ldots, x_n]$ und $[y_1, \ldots, y_m]$
Erzeuge Verkettung der Listen: $[x_1, \ldots, x_n, y_1, \ldots, y_m]$

Imperativ, z.B. Pascal:

```
TYPE list = ^listrec;
     listrec = RECORD elem: etype;
                      next: list
               END;

PROCEDURE append(VAR x: list; y:list);
BEGIN
   IF x = NIL
   THEN x := y
   ELSE append(x^.next, y)
END;
```

Anmerkungen:

- Implementierung der Listendefinition durch Zeiger
- Verwaltung von Listen durch explizite Zeigermanipulation
- Verkettung der Listen durch Seiteneffekt

Gefahren bei expliziter Speicherverwaltung:

- Zugriff auf Komponenten von NIL
- Speicherfreigabe von unreferenzierten Objekten:
  - keine Freigabe $\leadsto$ Speicherüberlauf
  - explizite Freigabe $\leadsto$ schwierig (Seiteneffekte!)

Deklarative Sprachen abstrahieren von Details der Speicherverwaltung (automatische Speicherbereinigung)
Obiges Beispiel in funktionaler Sprache:

```
data List = [] | etype : List

append([]  , ys) = ys
append(x:xs, ys) = x : append(xs,ys)
```

Diese Formulierung hat eine Reihe von Vorteilen:

- Diese Implementierung hat keine Seiteneffekte, wodurch man leichter die Korrektheit dieser Definition formal nachweisen kann.
- Die Korrektheit dieser Definition ist einsichtig wegen der einfachen Fallunterscheidung. Wir wollen z.B. Beweisen, dass folgende Aussage immer gilt:

  $\texttt{append}([x_1, \ldots, x_n], [y_1, \ldots, y_m]) = [x_1, \ldots, x_n, y_1, \ldots, y_m]$

  Wir beweisen dies durch Induktion über die Länge der 1. Liste:

  Ind. anfang: n=0: $\underbrace{\texttt{append}([], \underbrace{[y_1, \ldots, y_m]}_{ys}) = \underbrace{[y_1, \ldots, y_m]}_{ys}}_{\text{Anwenden der ersten Gleichung}}$

  Ind.schritt: $n > 0$:
  Hierzu nehmen wir an, dass die folgede Induktionsvoraussetzung gelte:

  $\texttt{append}([x_2, \ldots, x_n], [y_1, \ldots, y_m]) = [x_2, \ldots, x_n, y_1, \ldots, y_m]$

  Dann gilt:
  $\texttt{append}([x_1, \ldots, x_n], [y_1, \ldots, y_m])$
  $= \texttt{append}\ (x_1 : \underbrace{[x_2, \ldots, x_n]}_{xs}, \underbrace{[y_1, \ldots, y_m]}_{ys})$
  $= x_1 : \texttt{append}\ ([x_2, \ldots, x_n], [y_1, \ldots, y_m])$ (Anwendung der 2. Gleichung)
  $= x_1 : [x_2, \ldots, x_n, y_1, \ldots, y_m]$ (Ind. vor.)
  $= [x_1, \ldots, x_n, y_1, \ldots, y_m]$ (nach Def. von Listen)

- keine Zeigermanipulation
- keine explizite Speicherverwaltung
- Universelle Verwendung von `append` durch generische Typen ($\rightarrow$ *Polymorphismus*):

  ```
  data List a = [] | a : List a
  ```

  (`a` ist hierbei eine Typvariable)

  $\Rightarrow$ `append` auf Listen mit beliebigem Elementtyp anwendbar

Klassifikation von Programmiersprachen:

- *Imperative Sprachen:*
  - Variablen = Speicherzellen (veränderbar!)
  - Programm = Folge von Anweisungen (insbesondere: Zuweisung)
  - sequenzielle Abarbeitung, Seiteneffekte
- *Deklarative Sprachen*
  - Variablen = (unbekannte) Werte
  - Programm = Menge von Definitionen (+auszuwertender Ausdruck)
  - potenziell parallele Abarbeitung, seiteneffektfrei
  - automatische Speicherverwaltung
  - weitere Klassifikation nach der zu Grunde liegenden mathematischen Theorie:

    Funktionale Sprachen
    - Funktionen, λ-Kalkül [Church 40]
    - Reduktion von Ausdrücken
    - Funktionen höherer Ordnung
    - polymorphes Typkonzept
    - "Pattern Matching"
    - "Lazy Evaluation"

    Logiksprachen
    - Relationen, Prädikatenlogik
    - Resolution, Lösen von Ausdrücken [Robinson 65]
    - logische Variablen, partielle Datenstrukturen
    - Unifikation
    - Nichtdeterminismus

    Funktional-logische Sprachen: Kombination der beiden Klassen

Konkrete deklarative Sprachen:

- funktional:
  - LISP, Scheme: eher imperativ-funktional
  - (Standard) ML: funktional mit imperativen Konstrukten
  - Haskell: Standard im Bereich nicht-strikter funktionaler Sprachen
    $\rightarrow$ praktische Übungen
- logisch:

  - Prolog: ISO-Standard mit imperativen Konstrukten
  - CLP: Prolog mit Constraints, d.h. fest eingebauten Datentypen und Constraint Solver
- funktional-logisch:
  - Mercury (University of Melbourne): Prolog-Syntax, eingeschränkt
  - Oz (DFKI Saarbrücken): Logik-Syntax, nebenläufig
  - Curry: Haskell-Syntax, nebenläufig

Anwendungen deklarativer Sprachen

- prinzipiell überall, da berechnungsuniversell
- Aspekte der Softwareverbesserung:
  - Codereduktion ($\rightarrow$ Produktivität):
    - Assembler/Fortran $\approx 10/1$
    - keine wesentliche Reduktion bei anderen imperativen Sprachen
    - funktionale Sprachen/imperative Sprachen $\approx 1/(5-10)$
  - Wartbarkeit: lesbarere Programme
  - Wiederverwendbarkeit:
    - polymorphe Typen (vgl. `append`)
    - Funktionen höherer Ordnung $\rightsquigarrow$ Programmschemata

Anwendungsbereiche

- Telekommunikation:
  Erlang (nebenläufige funktionale Sprache):
  - ursprünglich: Programmierung von TK-Software für Vermittlungsstellen
  - Codereduktion gegenüber imperativen Sprachen: 10-20%
- wissensbasierte Systeme: Logiksprachen
- Planungsprobleme (Operations Research, Optimierung, Scheduling):
  CLP-Sprachen
- Transformationen: funktionale Sprachen, z.B. XSLT für XML-Transformationen

Literatur

Gute Einführungen in die funktionale Programmierung findet man in [Hudak 99, Thompson 96]. Ein Standardwerk zur logischen Programmierung mit Prolog ist [Sterling/Shapiro 94]. Überblicksartikel und Einführungen in die logisch-funktionale Programmierung findet man in [Antoy/Hanus 10, Hanus 94, Hanus 13].