Programmiersprachen

Vorlesung 6

Zusammenfassung

Georg Ernst Moser

Functional Programming

Imperative versus Functional Languages

Der Hauptunterschied zwischen imperativen und Funktionalen Sprachen ist das Fehlen der imperativen „Features“ in funktionalen Sprachen. Im Gegensatz, sind in den meisten imperativen auch Eigenschaften von funktionalen Sprachen integriert (Lambda Ausdrücke,..)

Eigenschaften von **imperativen Sprachen:**

- **Variablen**
- **desktruktive Assignements**
- **sequentielle Ausführung**: Sequentielle Ausführung ist zwar auch von der Art des Denkens in funktionalen Sprachen vorhanden, aber diese Sequenz kommt nicht daher das ein Befehl nach dem anderen ausgeführt wird. Es gibt keine Befehle, sondern nur Ausrücke (ein funktionales Programm ist ein großer Ausdruck). Die Reihenfolge (Sequenz) kommt daher, dass sie von uns künstlich über higher order functions, parameter (deren Wert vor Ausführung einer Funktion bekannt sein muss) eingeführt wird. In imperativen Sprachen ist die Reihenfolge direkt vorgegeben.
- Die Semantik in imperativen Sprachen spiegelt die Natur des Computers deutlich wieder.
- **Arbeiten mit Schleifen**

Eigenschaften von **funktionalen Sprachen:**

- Schleifen werden ersetzt durch:
  - Rekursion: gute funktionale Programmierer verwenden kaum Rekursion. Da fast alles mit higher order funktions gelöst werden kann.
  - Higher Order Functions (functional forms)
- In funktionalen Sprachen bastelst sich der Programmierer selbst Kontrollstrukturen. In Imperativen Sprachen sind diese in Form von if,while,... vorgegeben. Die Vorstellung, dass imperative Sprachen mit Schleifen und funktionale mit Rekursion arbeiten stimmt also nur bedingt.
- Programmieren ohne Seiteneffekte (auch keine Zuweisungen): Dadurch kann referentielle Transparenz erreicht werden. Dies bedeutet, dass gleiches immer durch gleiches ersetzt werden kann.
  Das aus imperativen (vor allem objektorientierten) Sprachen bekannte Problem „Wann ist etwas eine Kopie, das Original, eine Referenz,...“ ist dadurch nicht vorhanden.
  Der Unterschied zwischen einer echten Kopie und „potentieller Gleichheit“ (wofür eine Gleichheits- Funktion implementiert werden muss) bleibt bestehen.
- Datenobjekte und mathematische Funktionen bilden die Grundlage
  - symbolische Programmierung im Gegensatz zur Programmierung mit realen Zahlen
  - Mathematische Funktionen im Gegensatz zu Prozeduren (verhalten sich nur ähnlich wie echte Funktionen)

Funktionen

Eine Funktion ist eine Abbildung von einer **Domain** in einen **Range** (entsprechend einer Funktion in der Mathematik).
Die **Signatur** einer Funktion wird folgendermaßen angegeben;
name : domain → range (Bsp: suqare : integer → natural)

Um eine Funktion zu definieren bedarf es noch gewisser Regeln, die den Übergang eines Wertes von der Domain in den Range spezifizieren.
Bsp: square(n) ≡ n x n

Anwendung einer Funktion:
Bsp: square(2)

Auch in funktionalen Sprachen gibt es Variablen. Diese dürfen allerdings nur einmal zugewiesen werden (single assignment). Dies ist kein Nachteil, da beliebig viele solcher Variablen erstellt werden können (auch durch Generatoren).

Functional Forms (Higher Order Functions)

Im Detail kann man zwischen Functional Forms und Higher Order Functions unterscheiden, da mit Functional Forms aus den Einschränkungen des Typsystems ausgebrochen werden kann (Higher Order Functions sind immer an die Vorgaben der Sprache gebunden).

Functional Forms werden auf Funktion angewandt und liefern neue Funktionen als Ergebnis.
Ein bekannter Vertreter ist zum Beispiel der Kompositionsoperator:
$F \equiv G \circ H$ (G wird angewandt auf das Ergebnis der Anwendung von H auf ein Argument)
Ähnlich dem Punkt in Haskell: Wendet zuerst die rechte Funktion an und auf deren Rückgabe die linke Funktion.

Beispiele rekursiver Funktionen

```
n! ≡ if n = 0 then 1 else n ∗ (n-1)!

prime(n) ≡ if n = 2 then true else p(n, sqrt(n))

p(n, i) ≡ if (n mod i) = 0 then false
      else if i = 2 then true
            else p(n, i − 1)
```

Die Ersteller von funktionalen Sprachen waren meist Mathematiker. Diese bildeten wohl aus Gründen der Gewohnheit eine Syntax, welche den Programmierer mathematische Funktionen leicht definieren lässt.

Hilfsfunktionen werden eingesetzt wenn sich gewisse Funktionalität in einer Funktion alleine nicht schön abbilden lässt. Oft werden mit diesen auch Hilfsvariablen eingeführt.

Values, Bindings, Functions

Als Sprache wird in folgenden Beispielen ML verwendet. ML ist eine strikte Sprache (ungleich Haskell): es werden immer zuerst die Argumente ausgewertet, dann die eigentliche Funktion.

Variablenbindungen in ML:

```
val A = 3;
val B = ”a”;
```

A wird beispielsweise an 3 gebunden und kann somit nie für einen anderen Wert (z.B.: 5) stehen. Für die Bindung einer Variable an einen String,... gilt natürlich dasselbe.

Variablen können auch an Funktionen gebunden werden:

```
val sq = fn(x:int) => x * x;
```

Eine Definition über *fun* entspricht einer Zuweisung einer Funktion an eine Variable (mit Ausnahme einiger durch *fun* ermöglichter Vereinfachungen).

```
fun square(n:int) = n * n;
```

Funktionen können direkt auf einen Wert angewandt werden, ohne, dass ein Name für die Funktion vorhanden sein muss.

```
(fn(x:int) => x * x) 2
```

Ein Aufruf über den Namen ist selbstverständlich auch möglich:

```
2 * sq(A)
```

Factorial in C++ and ML

C++

Iterativ

```
int fact(int n) {
      int i = 1;
      for (int j = n; j > 1; j--)
            i = i * j;
      return i;
}
```

Rekursiv

```
int fact(int n) {
      if (n == 0)
            return 1;
      return n * fact(n - 1);
}
```

ML

```
fun fact(0) = 1
|      fact(n) = n * fact(n – 1);
```

An diesem Beispiel lässt sich die besonders kurze Syntax von funktionalen Sprachen erkennen. Ob das Schreiben von weniger Zeilen jedoch wirklich ein entscheidender Vorteil ist sei dahingestellt. Das Beispiel in C betreffend, würde wohl kein Programmierer die rekursive Variante wählen, obwohl diese Version der Routine kürzer ist. Die iterative Variante ist zwar syntaktisch länger, aber für die meisten einfacher verständlich. Kürze darf daher nicht mit Einfachheit gleichgestellt werden.
Anhand der obigen Beispiele kann beobachtet werden, dass der Vorteil der „Kürze“ von funktionalen Sprachen auch teilweise durch die andere „Art zu Programmieren“ zustande kommt.

Ein weiterer Grund für die kompakte Syntax in funktionalen Sprachen ist die Verwendung von Pattern Matching, welche das Aufspannen von großen Fallunterscheidungen unnötig macht. Technisch wäre es problemlos möglich diese Technik auch in imperative Sprachen zu integrieren, doch scheint dieses Konzept zu sehr ein Fremdkörper für „eingesessene“ Programmierer des imperativen Stils zu sein.

Expressions in the Lambda Calculus

Aus heutiger Sicht betrachtet bildet der Lambda Kalkül das Grundgerüst der funktionalen Programmierung.
Historisch betrachtet ist der Lambda Kalkül höchst wahrscheinlich nicht die Quelle der funktionalen Programmierung, sondern ein heute unbekannter Seitenpfad, der nicht weiter beachtet wurde.
Interessanterweise war der Lambda Kalkül in der Entwicklung von imperativen Sprachen sogar von größerer Bedeutung. So war er in der Sprache Algol 60 (entwickelt in den 50er Jahren) direkt enthalten. Doch die Verwendung des Lambda Kalkül in direkter Form harmonierte nur schlecht mit den imperativen Konzepten, mit welchen sie in Algol 60 gepaart wurde.
Nach einem langen Entwicklungsprozess wurde erkannt, dass die Art der Parameterübergabe welche hinter dem Lambda Kalkül steckt (call-by-name) nicht optimal ist.
Frühe funktionalen Sprachen, wie zum Beispiel LISP, setzten von Anfang an nicht auf den Lambda Kalkül. Bei ihnen kam als Übergabetechnik call-by-value zum Einsatz.

Der Lambda Kalkül ist an sich ein sehr einfacher Kalkül. Seine Ausdrücke sind folgendermaßen definiert:

- Die Grundelemente bilden Namen (Identifier), welche alles sein können (Zahlen, Symoble, …)
- Eine Funktionsabstraktion (ausgedrückt als Lambda)
  Nach dem Lambda steht ein oder mehrere Formale Parameter.
  Es folgt nach dem Punkt der Rumpf der Funktion
  Beispiel: (λx.x ∗ x) //Eine Funktion die x quadriert.
- Funktionsanwendungen: Links steht die Funktion, rechts der Wert auf den diese angewendet werden soll.
  Beispiel: ((λx.x ∗ x) 2)
  Prinzipiell kann alles auf alles angewandt werden. Zum Beispiel kann auch 2 auf 3 angewandt werden. Das Ergebnis ist allerdings „2 auf 3 angewandt“, da nichts definiert wurde.

Klammern sind nicht Teil der Syntax des Lambda Kalküls. Sie dienen lediglich dazu die Syntax klarer verständlich zu machen.
Die Anwendung geht von links nach rechts (keine Klammern notwendig)

Beispiele für Klammer-Äquivalenz:

```
e1 e2 e3 = ((e1 e2) e3)
λx.y z = (λx.(y z))          //Funktion geht bis zum Ende des Aufrufs
                             //(hört nicht auf wenn nicht geklammert)
```

Currying

Bei der Anwendung einer Funktion muss der Parameter nicht in Klammern gesetzt werden:
`f x == f(x)`

Der Lambda Kalkül unterstützt jedoch keine Funktionen mit mehr als einem Parameter. Folgende Funktion kann also nicht direkt ausgedrückt werden:
`f(x,y,z)`

Um diese Funktionalität trotzdem zu bieten wird **Currying** verwendet. Eine Funktion welche einen Parameter entgegennimmt, kann eine Funktion, welche einen weiteren Parameter entgegennimmt (x) zurückliefern. Dieser zurückgelieferten Funktion kann dann der nächste Parameter (y) übergeben werden.

Auswertung von
`(λx.λy.λz.x * y * z) 1 2 3`
Die Funktion wird angewandt auf die Argumente. Dementsprechend wird auch geklammert:
`((((λx.(λy.(λz.(x * y * z)))) 1) 2) 3)`
Der erste Parameter x wird durch das entsprechende Argument ersetzt:
`(((λy.(λz.(1 * y * z))) 2) 3)`
Der nächste Parameter wird ersetzt:
`((λz.(1 * 2 * z)) 3)`
Der letzte Parameter wird ersetzt:
`(1 * 2 * 3)`
Dies ist auch das Ergebnis der Berechnung, da im Lambda Kalkül weil die Multiplikation nicht definiert ist. Meistens wird in Programmiersprachen für solche Ausdrücke auch nicht der Lambda Kalkül verwendet (Definition solcher Operationen wäre hier sehr umständlich).

Der Lambda Kalkül eignet sich bestens dafür große Ausdrücke zu vereinfachen. Dies wird solange gemacht, bis primitive Funktionen vorliegen, welche ausgewertet werden müssen.

Substitution

Entspricht call-by-name.
Formal wird eine Ersetzung folgendermaßen ausgedrückt:
`[e1/x]e2`
„*e1* ersetzt alle freien (ungebundenen) Vorkommen von *x* in *e2* „
Ungebundene Vorkommen sind hierbei Namen, welche nicht durch ein Lambda gebunden werden.

Regeln:

```
[e/x1]x2    = e if x1 = x2
            = x2 if x1 ≠ x2

[e1/x1](λx2.e2)   = (λx2.e2) if x1 = x2
                        = (λx2.[e1/x1]e2) if x1 ≠ x2 and
                              x2 does not occur free in e1 (would be
                        bound → rename x2 to x3)
                        = (λx3.[e1/x1][x3/x2]e2) otherwise,
                              where x1 ≠ x3 ≠ x2 and
                              x3 does not occur free in e1 and e2

[e1/x](e2 e3) = ([e1/x]e2 [e1/x]e3)
```

Equivalence of Lambda Expressions

Im Lambda Kalkül gibt es nur 3 Konversions Regeln.

Umbenennen von Parametern (α conversion):
`λx1.e ↔ λx2.[x2/x1]e (where x2 is not free in e)`

Funktionsanwendung (β conversion):
`(λx.e1) e2 ↔ [e2/x]e1`

Eliminieren von redundanten Funktionen(η conversion). Wird eigentlich nie gebraucht, hilft jedoch dabei formal zu beweisen, dass der Kalkül vollständig ist.:
`λx.(e x) ↔ e (where x is not free in e)`

Dies heißt η Konversion, da zwischenzeitlich eine Gamma Regel eingeführt wurde, welche später jedoch wieder entfernt wurde.

```
λx.(e x)(f)       → e f       //η conversion
λx.(e x)(f)       → e f       //β conversion
λx.(e x)          → e         //η conversion → without Parameter
```

Reduction and Normal Form

Bei diesen Regeln werden die β und η Regeln von links nach rechts angewandt um Ausdrücke zu reduzieren.
Ausdrücke werden reduziert (vereinfacht), bis keine weitere Regel mehr anwendbar ist. Ist dies der Fall, so ist der Ausdruck in Normalform.
Jede mögliche Reihenfolge von Regelanwendungen, welche zu einer Normalform führt, führt zur selben Normalform.
Allerdings hat nicht jeder Lambda Ausdruck eine Normalform. Folglich besteht auch die Möglichkeit von Endlosberechnungen (eine Notwendigkeit für Turing Vollständigkeit),
Beim Anwenden von Reduktionen muss allerdings darauf geachtet werden dass diese „richtig, bzw. gerecht“ angewendet werden. Damit ist gemeint, dass nicht immer dieselbe Regel angewandt werden soll, welche zu keinem Ergebnis führt.

In **Haskell** funktioniert das Auswählen der anzuwendenden Regel folgendermaßen: Solange wie möglich wird gar keine Regel angewandt. Ist die Anwendung unumgänglich, so wird die Regel ausgewählt, die ein Ergebnis/Teilergebnis liefert, welches benötigt wird (=lazy Evaluation). Dies bedeutet nicht dass diese Anwendungsreihenfolge immer zu einem Ergebnis führt!

In **ML** wird erst immer das Argument ausgewertet und dann erst der Rest. Dies führt ebenfalls nicht immer zu einem Ergebnis.

Beispiel:

`(λx.(λy.x + y) 5) ((λy.y ∗ y) 6) =`
y wird durch 5 ersetzt.
`(λx.x + 5) ((λy.y ∗ y) 6) =`
Die Funktion *(λy.y ∗ y)* wird auf *6* angewandt.
`(λx.x + 5) (6 ∗ 6) =`
Die Funktion *λx.x* wird auf *(6x6)* angewandt.
`(6 ∗ 6) + 5`

Es existieren auch Varianten eines **typisierter** Lambda Kalküls:
Einfach typisierter Lambda Kalkül: Garantiert, dass jeder Lambda Ausdruck eine entsprechende Normalform hat (nicht mehr Turing vollständig!).
Erweiterter typisierter Lambda Kalkül: Gleiche Eigenschaften wie der normale Lambda Kalkül.

Lambda Ausdrücke in modernen Programmiersprachen:
Sind keine Lambda Ausdrücke im eigentlichen Sinn, sondern eigentlich ein „Closure“. In diesen dürfen keine ungebundenen Namen vorkommen (Namen entweder als Parameter oder durch die Umgebung gebunden,...)

LISP

Historisch ältester Vertreter der funktionalen Programmiersprachen. LISP ist eine symbolische Sprache, da deren Grundelemente sogenannte **Symbole** sind.

Symbole können sein:

- Atome - „A“, „AUSTRIA“
- Zahlen 68000

Weiters gibt es in LISP auch Listen. Diese werden dargestellt als geklammerte Ausdrücke, deren Elemente durch Leerzeichen getrennt sind.

```
(PLUS A B)
```

Listen können natürlich auch geschachtelt werden

```
((MEAT CHICKEN) WATER)
(UNC TRW SYNAPSE RIDGE HP)
```

Ursprünglich stand NIL für eine leere Liste. Aus Gründen der Lesbarkeit wurde aber auch () als gleichbedeutendes Atom eingeführt.

```
NIL and () represent the empty list
```

Atome repräsentieren gewisse Werte. Sie können also als Variablennamen gesehen werden. Zahlen repräsentieren ihren eigenen Wert (Zuweisungen wie „3 soll 5 enthalten“ sind nicht möglich).

Werte können an **Atome** durch **Zuweisungen** zugewiesen werden (imperative Methode):

Globale Zuweisung

```
(SETQ X (A B C))
```

Setze den Wert von X auf das Ergebnis der Auswertung von (A B C). (A B C) kann nämlich auch als Funktion gesehen werden: A wird angewandt auf B C.

Lokale Zuweisung

```
(LET ((X A) (Y B)) E)
```

Während der Auswertung von E gilt:
X steht für A
Y steht für B

Primitive Functions in LISP

Das erste Element einer Liste wird als Funktion interpretiert und auf den Rest der Liste angewandt. Manchmal will der Programmierer jedoch nicht, dass eine Liste als Funktion interpretiert wird (man will nur die Liste, oder das Atom selbst). Hierfür wird **QUOTE** verwendet:

```
(QUOTE A)                    steht für das Atom A selbst
'A                           gleichbedeutend mit (QUOTE A)
```

Funktionen:

Name	**Funktion**
CAR	Liefert erstes Element einer Liste. Auf einem Atom ist CAR nicht definiert.
CDR	Liefert den Rest (alles außer dem ersten Element) einer Liste
CONS	Steht für CONSTRUCT. CONS hat 2 Argumente. Es fügt das erste Argument als erstes Listenelement in das zweite Argument ein. Ist das erste Element eine Liste, fügt es die ganze Liste als erstes Element des zweiten Arguments ein.
ATOM	Ist das Argument ein Atom?
EQ	Vergleich auf Identität.
COND	Steht für CONDITIONAL. Argument ist eine Liste aus beliebig vielen Paaren. Der erste Teil der Paare ist eine Bedingung, welche ausgewertet wird. Der zweite Teil das Ergebnis welches zurückgelieferten wird, wenn die Bedingung zu TRUE evaluiert. COND gibt den Wert des ersten Paares zurück, dessen Bedingung zu TRUE evaluiert.
NULL	Ist Argument eine leere Liste?

Die Namen von CAR und CDR stammen von den Registern in denen die Funktionalität in frühen Versionen gespeichert war.

Beispiele:

```
(CAR ’(A B C)) = A
(CAR ’A) = error
(CDR ’(A B C)) = (B C)
(CDR ’(A)) = () = NIL
(CONS ’A ’(B C)) = (A B C)
(CONS ’(A) ’(B)) = ((A) B)
(ATOM ’A) = T
(ATOM ’(A)) = NIL                        Nil steht für false
(ATOM NIL) = T                           Nil ist Atom und Liste
(EQ ’A ’A) = T
(EQ ’A ’B) = NIL
(COND ((EQ ’X ’Y) ’B) (T ’C)) = C  Das letzte Paar hat als Bedingung T und
                                   entsprechicht so quasi einem else.
(NULL ’()) = T
```

Weiters existiert noch die Funktion ***Lambda***: Sie hat 2 Argumente.

1. Liste von formalen Parametern
2. Rumpf der Funktion die deklariert werden soll

Beispiel:

```
(LAMBDA (X Y) (PLUS X Y))          Lambdaausdruck der x und y addiert

((LAMBDA (X Y) (PLUS X Y)) 2 3)    → 2+3 = 5
```

DEFINE definiert den Namen einer Funktion. Namen können gleichzeitig für Funktionen, Atome,.. vergeben werden.
DEFINE nimmt als ersten Parameter den Namen der zu definierenden Funktion und als zweiten den zugehörigen Lambda Ausdruck entgegen.

```
(DEFINE (ADD (LAMBDA (X Y) (PLUS X Y))))
```

Beispiel: Definition der Funktion reverse, welche eine Liste “umdrehen” soll

```
(DEFINE (REVERSE (LAMBDA (L) (REV NIL L))))
(DEFINE (REV (LAMBDA (OUT IN)
      (COND ((NULL IN) OUT)
            (T (REV (CONS (CAR IN) OUT)
                  (CDR IN)))))))
```

In Lisp können Programmkonstrukte auch verwendet (im Code geschrieben) werden, wenn diese nicht nicht definiert wurden. Solange sie dies zur Ausführungszeit sind.
Im obigen Beispiel wird so die Hilfsfunktion REV erstellt.
Diese hat 2 Parameter
IN → Formaler Parameter der entgegengenommenen Liste
OUT → Formaler Parameter der zurückgelieferten Liste
Ist die entgegengenommenen Liste leer → fertig
sonnst → Rufe REV rekursiv mit OUT (an welches das erste Element von IN angehängt wird) und IN (welches um 1 gekürzt wird → der Rest von IN) auf.

In obigen Programm kann auch gesehen werden, dass in LISP oft viele Klammern zu schließen sind. Als Erleichterung dafür gibt es jedoch die eckige Klammer, welche alle noch offenen runden Klammern schließt.

Typische LISP Programmierer verwenden Higher Order Functions.
So wendet zum Beispiel „*MAPCAR SQUARE L*“ SUQARE auf alle Elemente der Liste L an.
In LISP gibt es viele solcher vordefinierten Higher Order Functions. Es ist aber auch einfach möglich solche selbst zu definieren.

Für LISP sind gigantische Funktionsbibliotheken vorhanden. Dies ist ein Grund dafür, das LISP sich in gewissen Kreisen immer noch hoher Beliebtheit erfreut. Vor allem für die Aufgaben von Biologen und Chemiker (DNA Sequenzen suchen,...) sind entsprechende Bibliotheken vorhanden. Dies wurden irgendwann in LISP geschrieben und niemals transferiert.

APL

APL (A Programming Language) ist eine imperative Programmiersprache, welche eine Zeit lang von IBM favorisiert wurde und über einen Bekanntheitsgrad verfügte, der dem von Java heutzutage entspricht.
In APL gibt es 2 Arten von Datenstrukturen:

- Skalare (Zahlen oder Characters)
- Arrays (Sequenzen von Skalaren, oder anderen Arrays)

Boolesche Werte sind nicht direkt vorhanden, sondern werden durch 0 (FALSE) und 1 (TRUE) repräsentieren.
Variablenzuweisungen erfolgen mit einem Pfeil (←)

```
X ← 123
X ← 'b'
X ← 56789
```

Zuweisung sind auch Ausdrücke und liefern dementsprechend Werte zurück:

```
X ← (Y ← 5 6 7 8 9) × (Z ← 9 9 7 6 5)
W ← Y − Z
```

Das × steht hier für den Kreuzproduktoperator. Das Array welches durch diese Operation entsteht wird an X zugewiesen.

In APL gibt es hunderte primitive Funktionen:
Zum Beispiel
Arithmetische Operationen: +, −, ×, ÷, | (rest of division)
Boolsche Operationen: ∧, ∨, ⍨, <, ≤, =, ≥, >, ≠

Im Programm werden diese Operationen auch genau in Form dieser Zeichen verwendet. Jeder Funktionsname besteht in APL aus einem Symbol.
Um dem APL Programmierer all diese Funktionssymbole, welche auf einer normalen Tastatur großteils nicht vorhanden sind, verfügbar zu machen, gab es eigene APL Tastaturen. Notfalls konnte auch eine Schablone auf die Tastatur gelegt werden.

Dies Art des Programmierens, wobei keine neuen Funktionen erstellt werden können, nennt sich **applikativer Programmierstil**. In modernen funktionalen Sprachen wird genau nach demselben Prinzip programmiert: Durch das anwenden von funktionalen Formen auf Funktionen werden neue Funktionen erzeugt (Lesbarkeit leidet allerdings).

Komplexere Fuktionen:

Symbol	Funktion	Beispiel
⍳ (i ohne Punkt)	Generiert Arrays	⍳5 ≡ 1 2 3 4 5
;	Hängt 2 Arrays zusammen	⍳4; ⍳5 ≡ 1 2 3 4 1 2 3 4 5
⍴	Arrangiert Arrays	2 3 ⍴ 1 2 3 4 5 6 ≡ 1 2 3 4 5 6 (erzeugt ein Array mit 2 Spalten und 3 reihen, welches mit den nachfolgenden Werten sequentiell befüllt wird)
/	Kondensiert Arrays Der linke Parameter ist ein Array mit Wahrheitswerten (eine Maske), welches genau gleich lange ist wie das rechte Array. Aus diesem werden alle Werte, welche im linken Array mit 1 maskiert wurden, in ein neues Array geschrieben	1 0 0 1 / ⍳4 ≡ 1 4

Funktionale Formen in APL:

In APL ist eine gewisse Form des Überladens integriert. Derselbe Operator kann je nach Kontext für verschiedene Operationen (funktionale Formen oder Funktionen) stehen.
So ist „/“ als funktionale Form zum Beispiel der Reduktionsoperator. Dieser wendet eine Funktion Schritt für schritt für alle Elemente eines Arrays an. Er nimmt also eine Funktion und ein Array entgegen.

```
+/(⍳3)            ≡ 6                          //(1+2+3)
+/(2 3 ⍴ ⍳6)      ≡ +/[2](2 3 ⍴ ⍳6) ≡ 6 15
+/[1](2 3 ⍴ ⍳6)   ≡ 5 7 9                      //das [1] wählt die 1. Dimension
```

Der innere-Produkt Operator wird als Punkt (.) dargestellt und wird in der Form *X f.g Y* angewandt, wobei *X* und *Y* Arrays und *f* und *g* Funktionen sind.
g wird elementweise auf die Zeilen von *X* angewandt. *f* wird anschließend spaltenweise auf das Ergebnis angewandt.
Ein Beispiel zur Anwendung wäre die Matrizen-Multiplikation:

```
X +.× Y
```

Der äußere-Produkt Operator, als Kugerl (∘) dargestellt , wird in der Form *A ∘.f B* angewandt, wobei *A* und *B* wieder Arrays und *f* eine Funktion ist. *f* wird auf jedes Paar aus Elementen von *A* und *B* angewandt.
Beispiel

```
1 2 ∘.× 3 4 5 ≡   3 4 5          (1x3 1x4 1x5)
                  6 8 10         (2x3 2x4 2x5)
```

Ein Beispielprogramm

Es soll ein Programm zur Berechnung von Primzahlen erstellt werden:

```
step 1:          (⍳N) ∘.| (⍳N)
```
(*⍳N*) ist das Array aller Zahlen bis n. Es wird der äußerer Produkt Operator mit dem “Rest der Division” auf 2 solcher Arrays angewandt. Es werden also alle Zahlen durch alle anderen Zahlen dividiert.
```
step 2:          0 = (⍳N) ∘.| (⍳N)
```
Es wird überprüft für welche Zahlen die Division 0 Rest ergeben hat. An diesen Stellen steht ein 1 für True
```
step 3:          +/[2] 0 = (⍳N) ∘.| (⍳N)
```
Die Anzahl der Zahlen mit 0 Rest wird aufsummiert.
```
step 4:          2 = (+/[2] 0 = (⍳N) ∘.| (⍳N))
```
Ist dies genau 2x der Fall, handelt es sich um eine Primzahl. Diese Zahlen werden mit 1 markiert, alle anderen mit 0
```
step 5:          (2 = (+/[2] 0 = (⍳N) ∘.| (⍳N))) / ⍳N
```
Anschließend wird der Kondensations-Operator angewandt. Da nur Primzahlen mit 1 markiert sind, werden nur diese ins Rückgabe-Array geschrieben. Schritt 5 ist das fertige Programm.

Obwohl APL in seiner ursprünglichen Form veraltet ist, gibt es von denselben Entwicklern einen neueren Ableger namens J.

ML

ML kann als Übergangssprache von den alten zu den neuen funktionalen Sprachen gesehen werden. Es gibt wesentlich mehr syntaktische Elemente:

Listen	[2, 3, 4]	
	[”a”, ”b”, ”c”]	(Strings)
	[true, false]	(boolsche Werte)
	[] is equivalent to nil	(nil wurde aus LISP übernommen)
Listen Operationen:	hd([1, 2, 3]) ≡ 1	(head)
	tl([1, 2, 3]) ≡ [2, 3]	(tail)
	1::[2, 3] ≡ [1, 2, 3]	(Konkatenation)
	[1, 2]@[3] ≡ [1, 2, 3]	(Zusammenhängen von Listen)

Lokale Variablenbindungen:

```
let val x = 5 in 2 * x * x end;
```

Lokale Variablenbindungen mit mehreren Ausdrücken:

```
local val x = 5 in val sq = x * x;
                   val cube = x * x * x
      end;
```

Im Gegensatz zu APL soll der Programmierer in ML die bestehenden Möglichkeiten erweitern, anstatt alleine mit diesen zu arbeiten.

Beispielprogramme:

```
fun reverse([]) = []
| reverse(x::xs) = reverse(xs)@[x];

fun insert(x, []) = [x]
| insert(x:int, y::ys) =
      if x < y then x::y::ys        --hänge x vorne an die Liste
            else y::insert(x, ys);  --füge in den rest der Liste ein
```

Der Kleiner Operator ist nur auf Zahlen (ganzen und Fließkomma) definiert. In ML kann jedoch nicht angegeben werden, dass die Funktion für Fließkommazahlen und ganze Zahlen verwendbar sein soll. Es gibt keine Typklassen. Für gewöhnlich werden die Typen durch Typinferenz bestimmt. Der Ausdruck *(x < y)* beinhaltet jedoch nicht genug Information um daraus ableiten zu können ob x und y Integer- oder Fließkommawerte sind. Deswegen muss hier immer explizit ein Typ angegeben werden.

```
fun sort([]) = []
| sort(s::xs) = insert(s, sort(xs));
```

Funktionen

In ML gibt es

anonyme Funktionen, Funktionen ohne Namen (Lambda Ausdrücke)

```
Definition:       fn(x:int) => −x;
                  fn(x:int, y:int) => x * y;
Anwendung:        val intnegate = fn(x:int) => −x;
Typ:              fn:int−>int
```

higher order functions:
In ML können im Gegensatz zu APL nicht eine Funktion und eine HOF mit dem selben Namen versehen werden, da Funktionen und HOF dasselbe sind und nicht unterschieden werden könnte.

```
Funktion:         fun compose(f,g) = f(g(x));
Typ:              fn:('a−>'b * 'c−>'a)−>('c−>'b)
```

curried Funktionen:

Funktion mit mehr als einem Argument werden entweder durch Tupel (ein Tupel als Eingangsparameter) oder currying realisiert.
Curried Funktionen werden in ML traditionell weniger häufig angewandt als in Haskell (Performance-Günde).

```
Funktion:         fun times(x:int)(y:int) = x * y;
Typ:              fn:int−>(int−>int)
Anwendung:        fun multby5 = times(5); −- eigenständige Funktion
```

Funktionale Formen

In ML wurde versucht den Programmieren weniger, aber besser strukturierte (leichter merkbare) Funktionale Formen zu bieten.

Die funktionale Form map entspricht dem MAPCAR aus LISP und benutzt currying.
`map length [[], [1, 2, 3], [3]] ≡ [0, 3, 1]`

fold entspricht dem Reduktions-Operator („/“) aus APL

```
fun fold(f, s, nil) = s
| fold(f, s, (h::t)) = f(h, fold(f, s, t));
use: fold((op+), 0, [1,2,3,4]) ≡ 10
```

Typen in ML

In ML wurde Typinferenz zum ersten mal eingesetzt.

predefined types: bool, int, real, string

lists: [1, 2, 3] : int list
[”a”, ”b”] : string list
nil : ’a list

tupels: (true, ”fact”, 7) : bool * string * int
(true, nil) : bool * (’a list)

records: {name = ”A”, id = 9} : {name:string, id:int}

functions: idfunction : ’a−>’a

type definitions: type intpair = int * int;
type ’a pair = ’a * ’a;
type boolpair = bool pair;

Mit Typdefinitionen werden lediglich neue Namen für etwas bereits vorhandenes eingeführt (ähnlich typedef in C++). Diese sind mit einem unbenannten äquivalenten Konstrukt kompatibel.

Daten in ML

„datatype“ entspricht dem „data“ in Haskell: Es werden neue Typen angelegt, deren Instanzen, ungleich zu Typdefinitionen mit „type“ nur untereinander kompatibel sind.

```
datatype color = red | green | blue;

datatype publications = nopubs | journal of int | conf of int;

datatype ’t stack = empty | push of ’t * ’t stack;
      values of stack:       empty
                             push(2, empty)
                             push(2, push(3, push(4, empty)))

datatype ’t list = nil | :: of ’t * ’t list;
```

Abstract Data Type in ML

Der interne Aufbau des abstrakten Datentyps *lifo* wird nach außen verschleiert.

```
abstype ’a lifo = Stack of ’a list
with exception error;

      val create = Stack nil;

      fun push(x, Stack xs) = Stack(x::xs);

      fun pop(Stack nil) = raise error
      |     pop(Stack(x::xs)) = Stack xs;

      fun top(Stack nil) = raise error
      |     top(Stack(x::xs)) = x;

      fun length(Stack nil) = 0
      |     length(Stack(x::xs)) = length(Stack xs) + 1;
end;
```

Module in ML

Im Gegensatz zu einem abstrakten Datentyp ist bei der Verwendung von „structure“ die Implementierung nach außen sichtbar.

```
structure S = struct
      exception error;
      datatype ’t Stack = ’t list;

      val create = Stack nil;

      fun push(x, Stack xs) = Stack(x::xs);

      fun pop(Stack nil) = raise error
      |     pop(Stack(x::xs)) = Stack xs;

      fun top(Stack nil) = raise error
      |     top(Stack(x::xs)) = x;

      fun length(Stack nil) = 0
      |     length(Stack(x::xs)) = length(Stack xs) + 1;
end;
```

Signature in ML

Ursprünglich wurden abstrakte Datentypen verwendet, falls der interne Aufbau nicht gezeigt werden sollte und Strukturen im anderen Fall.
Signaturen ermöglichen es auch eine „offene“ Struktur teilweise zu verstecken.

In diesem Beispiel wird Stack auf String spezifiziert.

```
signature stringStack = sig
      exception error;
      type string Stack;

      val create: string Stack;
      val push: string * string Stack -> string Stack;
      val pop: string Stack -> string Stack;
      val top: string Stack -> string;
end;
```