# Introduction (Fortran)

**Assembly Languages**

- Mnemonics instead of binary opcodes
- Reusable macros and subroutines
- A lot of instructions the programmer had to keep in mind

```
push ebp
mov ebp, esp
sub esp, 4
push edi
```

Drawbacks

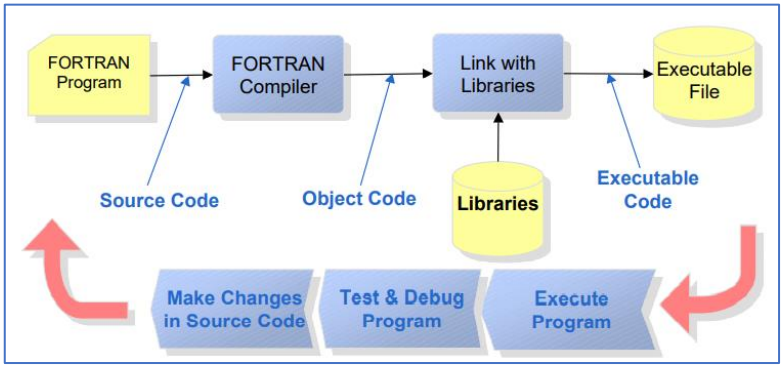
- Hardware specific
- Large set of minutiae instructions
- Dozens of instructions for simple expressions $(x^2 + y^2)$

**Programming Languages**

| | |
|---|---|
| High Level | Fortran |
| Structured | ALGOL |
| Functional | Lisp |
| Logical | Prolog |
| Domain Specific | COBOL |
| Declarative | SGML |
| Object Oriented | Smalltalk |
| Parallel | Occam |
| Modular | MODULA-2 |
| Mixed | C#, Java |

**Fortran** (**For**mula **Tran**slation)

- First high level programming language
- Still in use for supercomputers
- Still in use for numeric and scientific computations
- Compiled language
- *Implicit none* to enforce variable typing

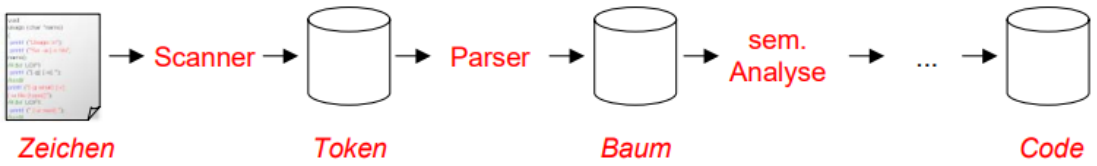



```
program ConvertF2C
  implicit none
  ! ------------------------------------------- Declare
  real*8 :: tempC, tempF, FACTOR
  character*32 :: arg
  integer*4 :: ZERO_SHIFT
  parameter (ZERO_SHIFT = 32, FACTOR = 5./9.)

  ! ------------------------------------------------Input
  print*, "Enter in Fahrenheit ..."
  read*, tempF

  ! ------------------------------------------- Compute
  tempC = FACTOR * (tempF - ZERO_SHIFT)
  ! ------------------------------------------- Output
  print*, "The corresponding Centigrade temp is "
  print*, tempC, " degrees."
end
```

# Compiler (Java) - Übersicht

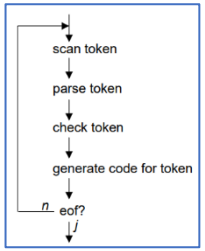
**Dynamische Struktur eines Compilers**

1. Zeichenstrom (Lexikalische Analyse = Scanning)
2. Tokenstrom (Syntaxanalyse = Parsing)
3. Syntaxbaum (Semantische Analyse = Typprüfung,…)
4. Zwischensprache (Optimierung, Codeerzeugung)
5. Maschinencode




**Einpass-Compiler**

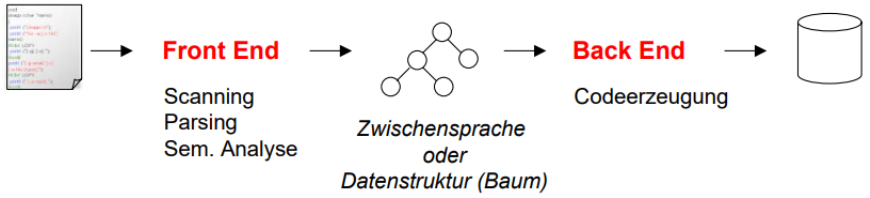
Während das Quellprogramm gelesen wird, wird bereits das Zielprogramm (Code) erzeugt. Die einzelnen Phasen arbeiten verzahnt.




- Schnell, einfach
- Kann nicht voraussehen

**Zweipass-Compiler**

- Langsamer
- mehr Speicherverbrauch
- Bessere Portierbarkeit
- Kombination beliebiger Front- und Back Ends möglich
- Zwischensprach ist einfacher optimierbar als Quellsprache

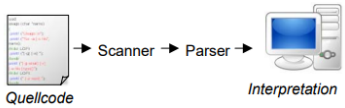



**Compiler VS Interpreter**

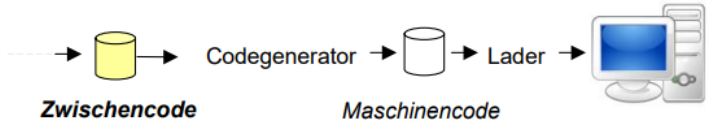
*Compiler* übersetzt in Maschinencode

*Interpreter* führt Quellprogramm «direkt» aus

- Kein Maschinencode
- Langsamer als Compilation

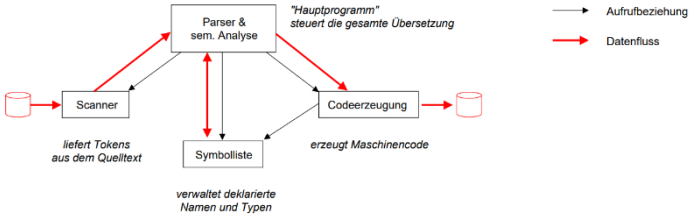



**Just In Time Compilation** (JIT)




- All in one — Gesamter Code beim Start übersetzt
- Incremental — Vor der ersten Ausführung übersetzt
- Hot-spot — Code wird interpretiert. Häufig durchlaufene Teile werden in Maschinencode übersetzt.

**Statische Struktur eines Compilers**



# Compiler - Grammatik

## Grammatik

- TS Terminalsymbole — Werden nicht mehr weiter zerlegt
- NTS Nonterminalsymbole — Werden weiter zerlegt
- Produktionen — Ableitungsregeln
- Startsymbol — Oberstes Nonterminal-Symbol

## Rekursionen

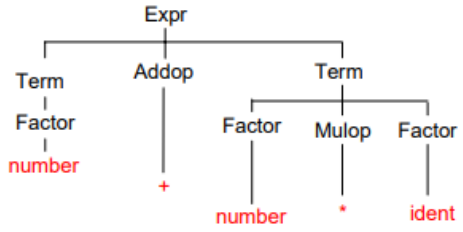
- Linksrekursion $A = b|Aa$ $A \to Aa \to Aaa$
- Rechtsrekursion $A = b|aA$ $A \to aA \to aaA$
- Zentralrekursion $A = b|$ "(" $A$ ")" $A \to (A) \to ((A))$

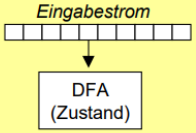
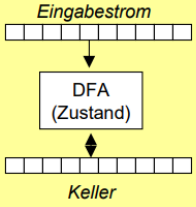
## EBNF – Schreibweise

| *Symbol* | *Bedeutung* | *Beispiele* |
|---|---|---|
| = | Trennt Regelseiten | $A = b\ c\ d.$ |
| . | Schliesst eine Regel ab | $A = b\ c\ d.$ |
| \| | Trennt Alternativen | $a \mid b \mid c \quad \to a, b\ or\ c$ |
| (...) | Fasst Alternativen zusammen | $a\ (b \mid c) \quad \to ab \mid ac$ |
| [...] | $0-1$ Wiederholungen | $[a]\ b \quad \to ab \mid b$ |
| {...} | $0-n$ Wiederholungen | $\{a\}\ b \quad \to b \mid ab \mid aab \mid \ldots$ |

## Abstrakter Syntaxbaum (AST)

- z.B. für $10 + 3 \cdot i$




| | Reguläre Grammatiken | Kontextfreie Grammatiken |
|---|---|---|
| **Anwendung** | Lexikalische Analyse | Syntaxanalyse |
| **Erkennung** | **DFA** (kein Keller)  | **PDA** (Keller)  |
| **Produktionen** | A = a \| b C. | A = a. |
| **Probleme** | Klammerkonstrukte | Kontextsensitive Konstrukte (z.B. Typprüfungen, ...) |





## Reguläre Grammatik / DEA = DFA

Eine Grammatik heisst *regülar*, wenn sie sich durch Regeln der folgenden Art ausdrücken lässt.

- $A = a \qquad a, b \in TS$
- $A = b\ B \qquad A, B \in NTS$

Lässt sich durch eine einzige Regel ohne Rekursion ausdrücken.

## Kontextfreie Grammatik / Kellerautomaten (PDA)

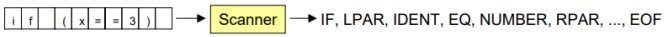
Eine Grammatik heisst *kontextfrei*, wenn alle Produktionen folgende Form haben

- $A = a. \qquad A \in NTS \quad a \in \{TS, NTS\}$

# Compiler (Java, Wasm)

**Lexikalische Analyse** (Nicht Teil der Syntaxanalyse) = Aufwendig!

1. Liefert Terminalsymbole (Tokens)




*Zeichenfolge* — *Tokenfolge (muss mit EOF enden)*

2. Überliest bedeutungslose Zeichen
   - Leerzeichen
   - Tabulatoren
   - Zeilenenden
   - Kommentare

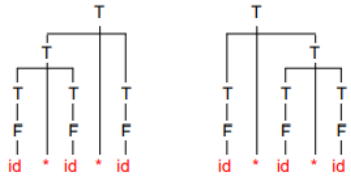
*Mehrdeutigkeit*

Eine Grammatik ist mehrdeutig, wenn man für einen Satz mehrere Syntaxbäume angeben kann.

- Ungeeignet zur Syntaxanalyse

T = F | T "*" T.
F = id.

Beispielsatz: id * id * id



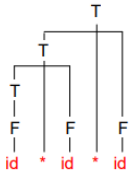




*Beseitigung von Mehrdeutigkeit*

**Die Grammatik kann umgeformt werden zu**

T = F | T "*" F.
F = id.




d.h. T hat Priorität vor F

nur dieser Syntaxbaum ist möglich

**Scanner** (Lexikalische Analyse)

- Zeichen lesen
- Es gibt nur einen einzigen Scanner pro Compiler
- Es wird ein Symbol vorausgeschaut

**Parser** (Syntaxanalyse)

- Zeichen in Tokens umwandeln
- Eine Methode für jede Produktion

**Syntaxanalyse**

- Reguläre Grammatiken

**Codeerzeugung**

**Web Assembly Instructions** (wasm)

- WASM is a *Compiler Target* (code generated by compilers)
- Applications can be compiled to WASM and run in browser

```
static void factor() throws Exception {
        if (Scanner.la == Token.LBRACK) {
                Scanner.scan();
                expr();
                Scanner.check(Token.RBRACK);
        } else if (Scanner.la == Token.NUMBER) {
                Scanner.scan();
        }
}
```

# Logik Programmierung (Prolog)

**Wissensrepräsentation** (WR)

Formalismus zur Darstellung von Wissen über einen bestimmten Bereich. Eine *formalisierte Repräsentation* ermöglicht es, dass neues Wissen aus bestehendem Wissen mittels einer «Logikmaschine» *Inferenzmaschine* abgeleitet werden kann.

**Formeln**

| | | | |
|---|---|---|---|
| Doppelte Negation | $\neg(\neg A)$ | $\Leftrightarrow$ | $A$ |
| Implikation | $A \rightarrow B$ | $\Leftrightarrow$ | $(\neg A \vee B)$ |
| De Morgan | $\neg(A \wedge B)$ | $\Leftrightarrow$ | $(\neg A \vee \neg B)$ |
| Distributivität | $C \vee (A \wedge B)$ | $\Leftrightarrow$ | $(C \vee A) \wedge (C \vee B)$ |
| Assoziativität | $(A \wedge B) \wedge C$ | $\Leftrightarrow$ | $A \wedge (B \wedge C)$ |
| Äquivalenz | $A \Leftrightarrow B$ | $\Leftrightarrow$ | $(A \rightarrow B) \wedge (B \rightarrow A)$ |

**Prädikatenlogik**

Erweiterung der Ausdrucksmächtigkeit durch Hinzunahme von

- Prädikaten Boolsche Funktion: $p(x_i)$
- Variablen $x_i$
- Quantoren $\forall, \exists$

**Quantor Regeln**

| | | |
|---|---|---|
| Vertauschungsregel | $\forall x\ A(x)$ | $\neg\exists x\ \neg A(x)$ |
| | $\forall x \in K\ A(x)$ | $\neg\exists x \in K\ \neg A(x)$ |
| Allquantor | $\forall x \in K\ A(x)$ | $\forall x(x \in K \Rightarrow\ A(x))$ |
| Existenzquantor | $\exists x \in K\ A(x)$ | $\exists x(x \in K \wedge\ A(x))$ |

**Allgemeingültigkeit, Erfüllbarkeit und Inferenz**

- Logische Folgerung $M \vDash \gamma$
- Syntaktische Umformung $M \vdash \gamma$

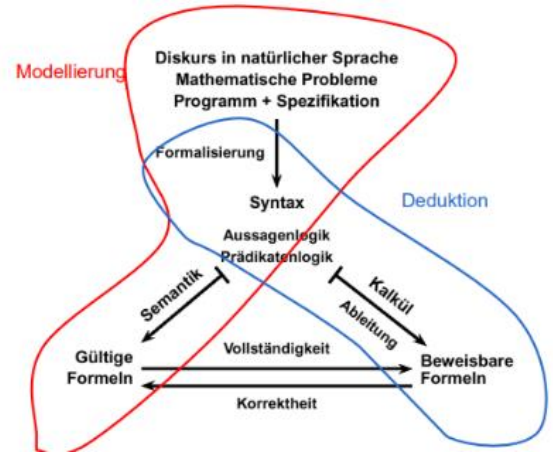
**Aussagenlogik**

- Disjunktive Normalform DNF
  $(L_{1,1} \wedge L_{1,2} \wedge L_{1,3}) \vee (L_{2,1} \wedge L_{2,2} \wedge L_{2,3}) \vee (L_{3,1} \wedge L_{3,2} \wedge L_{3,3})$
- Konjunktive Normalform KNF
  $(L_{1,1} \vee L_{1,2} \vee L_{1,3}) \wedge (L_{2,1} \vee L_{2,2} \vee L_{2,3}) \wedge (L_{3,1} \vee L_{3,2} \vee L_{3,3})$

**Aussagenlogik - Eigenschaften**

- Monoton
- Einschränkung der Modelierung auf *Fakten* bzw. *Klauseln* (True / False)
- Einschränkung der Ausdrucksmächtigkeit: *Keine Quantoren*

Falls WB1 |= α , dann auch (WB1 ∪ WB2) |= α



# Logik Programmierung (Prolog)

**Hornklausel** (Disjunktionsterm)

- Disjunktion von Literalen $\Phi_1 \vee \Phi_2 \vee \Phi_3 \dots \vee \Phi_n$
- $\Phi_i$ ist ein atomarer Ausdruck oder ein negierter atomarer Ausdruck
- Eine Hornklausel ist eine Klausel mit höchstens einem positiven Literal

$$\neg p \vee \neg v \vee \dots \vee \neg t \vee u$$

**Definite Hornklausel = Regel**

- Genau ein positives Literal $\neg x_1 \vee \dots \vee \neg x_n \vee y$
- Als Implikation dargestellt $x_1 \wedge \dots \wedge x_n \rightarrow y$
- Wenn $x_1, \dots, x_n$ wahr sind, dann ist auch $y$ wahr

**Zusammenfassung: Hornklausel**

Mit der Hornklausel lassen sich Wissensbasen aufbauen, mittels

- Fakten
- Regeln
- Variablen

Anfragen an die Wissensbasis werden als spezielle Klauseln formuliert und mittels eines linearen Resolutionsbeweises beantwortet.

Nachteil:

- Weniger ausdrucksstark als Prädikatenlogik 1. Stufe.

**Prolog**

- Basiert auf Hornklausel und linearem Resolutionsbeweis

**Faktenklausel = Fakt**

- Keine negativen Literale $y$
- Als Implikation dargestellt $1 \rightarrow y$
- $y$ ist wahr

**Zielklausel = Anfrage**

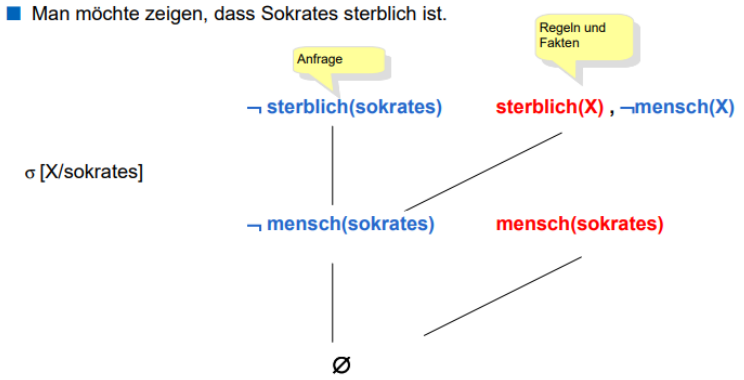
- Kein positives Literal $\neg x_1 \vee \dots \vee \neg x_n$
- Als Implikation $x_1 \wedge \dots \wedge x_n \rightarrow 0$
- In Worten $x_1, \dots, x_n$ sind nicht alle wahr

**Negation as Failure**

- Interpretation der Hornklausel

**Ordered Linear Resolution for Definite Clauses (OLD)**

- Starte mit einer negierten Anfrage als $C_0$
- Lineare Resolution: nimm im nächsten Resolutionsschritt die Resolvente $C_i$ al seine der beiden Klauseln und resolviere mit geeigneter Klausel $S_i$.



# Logik Programmierung (Prolog)

**Prolog** (**Pro**gramming **Log**ic) – Basics

- UND ∧ wird zu ,
- ODER ∨ wird zu ;
- Implikation ← wird zu :-
- Jede Klausel wird mit einem Punkt «.» beendet
- Variablen: Grossbuchstaben
- Funktionen: Kleinbuchstaben
- Eine beliebige Unifikation mit «_»
- Negation mit *not(...)*

Beispiele – Regeln

- $\forall x, y\ (g(x,y) \wedge m(x) \rightarrow s(x,y))$ schwester(X, Y) : − geschwister(X, Y) , mann(X).
- $\forall x, y\ (v(x,y) \vee m(x,y) \rightarrow e(x,y))$ elternteil(X, Y) : − mutter(X, Y) ; vater(X, Y).
- Alle deren Vater bekannt ist kennt_vater(X) : − vater(_, X).
- Negation von *older* younger(X, Y) : − not(older(X, Y))

Beispiele – Fakten

- Prädikat: *girl()* girl(susan).
- Prädikat: *likes()* likes(susan, cigarettes).

**Arithmetik**

- Zuweisung eines Wertes is
- Änderung eines Wertes + - * /
- Vergleich eines Wertes = \= >= < =<

**Listen und Funktionen / Operationen**

- Liste zuweisen X = [...]
- Liste als Argument p([...])
- Restlistenoperator | [Head | Tail]
- X ist in Liste L member(X, L)
- LN = L1 + L2 append(L1, L2, LN)
- Summe aller Werte sumlist(L, S)
- Alle Lösungen als Set setof(...)

Beispiele – Arithmetik

- $X = 5 \wedge Y = 8 \wedge Z = X + Y$ X is 5 , Y is 8 , Z is X + Y.
- Addition als Prädikat *plus()* plus(X, Y, Z) : − Z is X + Y.

Beispiele – Listen

- Summe S von [2,3,4] sumlist([2,3,4], S).
- Ist 3 in der Liste [2,3,4] member(3, [2,3,4]).
- Y = X + X append(X, X, Y).
- Alle Kinder von Tina setof(X, mutter(tina, X), S)

| | |
|---|---|
| append(List1, List2, List3) | fügt Liste1 und Liste2 zu Liste3 zusammen |
| member(Element,List) | Element ist in Liste enthalten |
| delete(Element,List1,List2) | Lösche alle übereinstimmenden Elemente aus Liste |
| flatten(List1,List2) | Umformung von geschachtelter Liste in flache |
| set(List1, List2) | Alle Duplikate entfernen |
| permutation(List1, List2) | Erzeuge Permutation von Liste |
| reverse(List1, List2) | Liste umkehren |
| last(list,X) | letztes Element aus Liste |
| subtract(List1,List2,List3) | Lösche Element aus Liste2 die in List1 enthalten sind |
| sumlist(List, N) | Addiere alle Zahlen in Liste |
| union(List1,List2, List3) | Vereinigung zweier Listen |
| setof(...) | Alle Lösungen als Set (ohne Duplikate) |
| bagof(...) | Alle Lösungen als Bag (mit Duplikaten) |

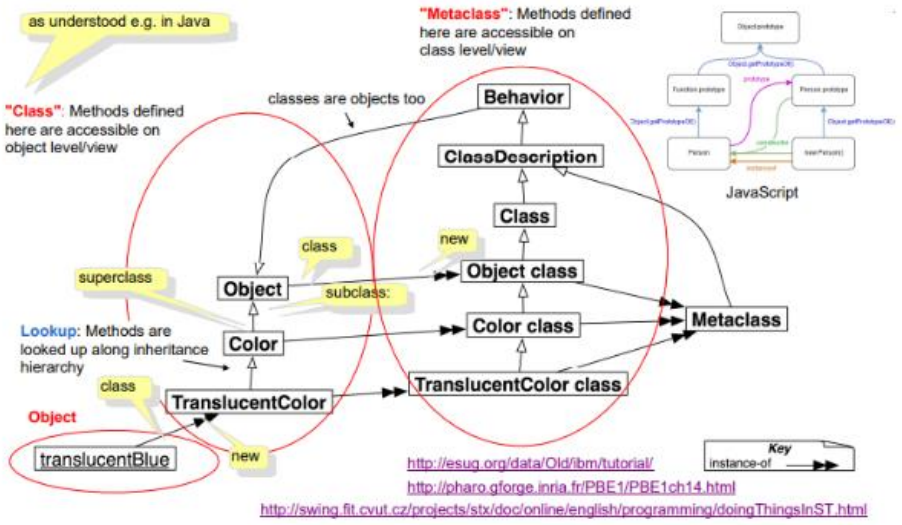
# Smalltalk (Objekt Orientiertes Programmieren)

**Smalltalk programming language**

- *Everything is an object* or a message
- Everything happens by *sending messages to objects*
- All the code is available, readable and changeable at runtime

**Smalltalk Object Model**

- Everything is an object
  - Things only happen by message passing
  - Variables are dynamically typed
  - Classes are also objects
- State is private to objects
- (All) Methods are public
- (Nearly) every object is a reference
- Single inheritance




**Smalltalk Syntax**

- Variablen Deklarieren | $x\ y$ |.
- Zeichen Instanzieren $A.
- Kommentare "This is a comment"

Vergleichs-Operationen

- true if the *objects* are the same ==
- true if the *values* are the same =

Strings

- Zuweisung $x \coloneqq "Hello\ World"$.
- Konkatinieren 'Hello', ' World'

Arrays

- Array: 1,2,3 #(1 2 3).
- Array-Länge #(1 2 3) size.

Blöcke

- Ananomy Methoden [...].
- If-Abfrage (true) ifTrue: [...].
- Loop von $s$ nach $e$ ... s to: e do: [: i |... |cr].

Boolean

- Und- / Oder- Verknüpfung ... & ... / ... | ...

Zu beachten

- Alle Ausdrücke enden mit einem Punkt «.»!
- Mathematische Ausdrücke werden von Links nach Rechts ausgewertet!

# Modula (Modulare Programmierung)

**Algol**

- Sprache um Algorithmen zu beschreiben
- Ausführbar auf Computern
- Schwierig zu Parsen (Compiler schreiben)

**Pascal**

- Systematisch, Effizienz
- Weiterentwicklung von Algol

**Modula-2**

- Modularisierung und getrennte Kompilierung von Programmen
- Verwendung von Bibliotheken (Imports)
- Geeignet für grosse Projekte

**Konzepte von Modula-2**

- *Information Hiding*: Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessern über definierte Schnittstellen.
- *Sicherheit durch Typisierung:* Die Programmierfehler vermeiden, die entstehen können, wenn der Compiler gefährliche, implizite Typen-Konvertierungen durchführt.

**Prozeduren und Coroutinen**

- Coroutinen können ihren Ablauf unterbrechen und später wieder aufnehmen, wobei sie ihren Status beibehalten.


**Java / OOP**

- Encapsulation of data and functionality
- Seperation of concerns
- Inheritance for reuse
- Package as a container

**Limits of OOP**

- Complexity grows
- Objects are interconnected

**Java ist Modular**

- Pckages as containers for classes
- Jar Files to package classes

**Was fehlt in Java?**

- Klar spezifizierte Abhängigkeiten
- Klar definierte Sichtbarkeiten
- Versionierung der Module

**Ein Modul in Java 9**

# Funktionale Programmierung (Lisp) - Basics

**Lisp** (LISt Processor) Eigenschaften

- Unterstützt funktionale Programmierung
- Nicht rein funktional: Multiparadigmensprache
- Bessere Sprachen für funktionale Programmierung, z.B. Haskell
- Dynamisches Typenkonzept

**Lisp Konstrukte**

- Quote — Nicht evaluieren
- Funktionen — Definition mit *defun*
- Variablen — *Set* bindet Werte an Symbole
- Konstenten — *t* = TRUE, *nil* = NULL
- Logik Operatoren — or, and, not, equal

```
;; Präfix-Notation
(+ 1 2 3 4)

;; Zahlen und Strings evaluieren sich selbst
3.14
"Hallo Lisp"

;; Funktions - Definition
;; (defun fname (params) (body))
(defun square (n) (* n n))
;; Funktions - Aufruf
;; (fname param) oder (funcall 'fname param)
(square 4)

;; Befehle:
;; cons = Liste konstruieren
(cons 1 '(2 3 4 5))
;; car  = Erstes Element einer Liste
(car '(1 2 3 4 5))
;; cdr  = Restliche Elemente einer Liste
(cdr '(1 2 3 4 5))
```

```
;; Variablen - Definition
(set 'abc '(a b c))
;; Variablen - Evaluierung
abc

;; IF: Bedinge Evaluation
(if nil          ;; IF
    "Then"       ;; THEN
    "Else")      ;; ELSE

;; COND: Bedinge Evaluation
(cond
    ((= a 0) "zero")         ;; CASE
    ((> a 0) "positive")     ;; CASE
    (t       "negative"))    ;; DEFAULT
```

```
;; Optional - Parameter (Ohne Default)
(defun opt (a b &optional c)
    (list a b c))

;; Optional - Parameter (Mit Default)
(defun opt (a b &optional (c 99))
    (list a b c))

;; Rest - Parameter
;; Restliche Parameter als Liste
(defun rst (a b &rest c)
    (list a b c))

;; Key - Parameter
(defun beispiel (&key a b c)
    (list a b c))
;; Aufruf
(beispiel :b 1 :a 2 :c 5)
```

# Funktionale Programmierung (Lisp)

A **closure** is a function or reference to a function together with a referencing environment – a table storing a reference to each of the non-local variables of that function.

- Closures ermöglichen es, Funktionen mit einem Zustand zu versehen.

**Pure Funktionen**

- Funktionsergebnis hängt ausschliesslich von den übergebenen Argumenten ab.
- Programmieren ohne Seiteneffekte

**Rekursive Funktionen**

Summe einer Liste

```
(defun list-sum(l)
    (if (null l) 0
        ;; else
        (+ (first l) (list-sum (rest l)))
    )
)
```

Funktion äquivalent zu mapcar

```
(defun map-list (f l)
    (if (null l) nil
        ;;else
        (cons (funcall f (first l))
              (map-list f (rest l)))))
```

**Prozeduraler Ansatz** Prozeduren, Funktionen

**OO Ansatz** Objekte, Klassen

- Java
- Zustand des Objekts in Instanzvariablen
- Methoden zum Ändern des Zustands

**Funktionaler Ansatz** Funktionen

- Lambdas
- Closures
- Funktionen höherer Ordnung

### Object-Oriented

- Data and the operations upon it are tightly coupled
- Objects hide their implementation of operations from other objects via their interfaces
- The central model for abstraction is the data itself
- The central activity is composing new objects and extending existing objects by adding new methods to them

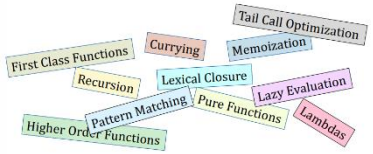
### Functional

- Data is only loosely coupled to functions
- Functions hide their implementation, and the language's abstractions speak to functions and the way they are combined or expressed
- The central model for abstraction is the function, not the data structure.
- The central activity is writing new functions

# Funktionale Programmierung (Lisp)

**Definition**

Keine einfache, allgemeine akzeptiert Definition (Sammlung von Merkmalen).




**Lambda-Kalkült und Turingmaschine**

Mit dem *Lambda-Kalkül* ($\lambda$) lässt sich alles ausdrücken, was man mit einer *Turingmaschine* ausdrücken kann. Das *Lambda-Kalkül* ist die theoretische Basis der funktionalen Programmierung. Moderne funktionale Programmiersprachen sind z.B. *ML* oder *Haskell*.

**Lambda-Kalkül – Formal**

- Funktionsabstraktion $\lambda x.A$ anonyme Funktion
- Funktionsapplikation $F\ A$ Funktion $F$ angewendet auf $A$
- Identität $\lambda x.x$

**Eigenschaften**

- Ähnlichkeiten zu SQL
- Konsistente Datenstrukturen
- Kombinierbar zu beliebigen Anfragen

**Funktionen höherer Ordnung**

- Funktionen als Parameter von Funktionen und / oder Funktionen als Rückgabewert von Funktionen.

**Funktionen mit Listen**

- Filter *(filter-list (lambda (n) (> n 10)) '(4 23 1 12 -22 73))*
- Reduce *(reduce-list #'* 0 '(1 2 3 4 5)*

```
;; Lambda-Kalkül
(lambda (n) n)

;; Liste filtern
(defun filter-list (f seq)
    (cond ((null seq) nil)
        ((funcall f (car seq))
            (cons (car seq) (filter-list f (cdr seq))))
        (t (filter-list f (cdr seq)))))

;; Beispiel = (23 12 73)
(print(filter-list (lambda (n) (> n 10)) '(4 23 1 12 -22 73)))

;; Liste reduzieren
(defun reduce-list (f init seq)
    (cond ((null seq) init)
        (t (funcall f (car seq) (reduce-list f init (cdr seq))))))

;; Beispiel = 120
(reduce-list #'* 1 '(1 2 3 4 5))
```

# Funktionale Programmierung (Lisp)

**Partielle Anwendung und Currying**

- Partielle Anwendung — Festlegen eines Teils der eigentlich benötigten Argumente einer Funktion $f$, die eigentlich $n$ Argumente erwartet.
- Currying — Umwandlung einer Funktion mit *mehreren Argumenten* in eine Funktion mit *einem Argument*.
- Komposition — Ausgabe einer Funktion ist die Eingabe einer anderen Funktion.
- Pipeline — Mehrere Funktionen hintereinanderschalten.

**Partielle Anwendung**

$$f: (X, Y, Z) \to N, \qquad partial(f): (X, Y) \to N$$

```
(defun partial (f &rest args)
    (lambda (&rest more-args)
        (apply f (append args more-args))))

(setfun sum-list
        (partial #'reduce-list #'+ 0))

(sum-list '(1 2 3 4)) ; = 10
```

**Komposition**

$$f \circ g$$

```
(defun times2 (x) (+ x x) )

(defun compose-simple (f g)
    (lambda (&rest args) (funcall f (apply g args))))

(print(funcall (compose-simple #'times2 #'times2) 5)) ; = 20
(print(funcall (compose-simple #'times2 #'+) 3 4)) ; = 14
```

**Currying**

$$f(a, b) \to h(a)(b) \equiv curry(f)$$

```
(defun curry2r (f)
    (lambda (b)
    (lambda (a)
        (funcall f a b))))

(setfun gt (curry2r #'>)) ;(funcall (gt 2) 3) = True
(setfun lt (curry2r #'<)) ;(funcall (lt 2) 1) = False
(setfun in-range (and (gt 5) (lt 15)))
(in-range 10) ; True
```

**Pipeline**

$$f_1 \circ f_2 \circ \ldots \circ f_n$$

```
(defun pipeline (&rest funcs)
    (lambda (&rest args)
        (first (reduce
            (lambda (ar f) (list (apply f ar)))
            funcs
            :initial-value args))))

(setfun nostring (pipeline #'stringp #'not))
(print (nostring "asdf")) ; = NIL
```

# Funktionale Programmierung (Python)

**Returnwert**

- Funktionen ohne *return* liefern None zurück
- Mehrere Werte können als *Tupel* übergeben werden

```
def func(x):
    return # None

def func(a, b, c):
    return a, b, c # Tuple
```

**Default - Parameter**

```
# Function with defaults
def func(a, b=2, c=3):
    print (a, b, c)

# (1, 2, 3)
func(1, 2)
# (1, 2, 3)
func(1, b=2, c=3)
```

**Default - Parameter**

```
# Function with key and rest
def func(*rest, **key):
    print (rest, key)

# (1, 2, 3)
func(1, 2, 3)
# ((1, 2), {'a':3, 'b':4, 'c':5})
func(1, 2, a=3, b=4, c=5)
```

**Lambda Ausdrücke**

- Lambda-Ausdrücke können anstelle von Funktionen verwendet werden.

```
func = lambda x: print(x) # Lambda function
```

**Funktionen höherer Ordnung**

```
def splat (f):
    def fs (seq):
        return f(*seq)
    return fs

def unsplat (fs):
    def f (*args):
        return fs(args)
    return f

add1 = splat(add)

add(1, 2)    # 3
add1([1, 2]) # 3
```

**Funktionsanwendungen**

- *map* und *filter* liefern Iteratoren
- *reduce* liefert ein Resultat

```
list(map(abs, [1, -2, 3, -4])) # [1, 2, 3, 4]
list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4])) # [2, 4]
functools.reduce(lambda a, b: a + b, [1, 3, 5, 6, 2]) # 17
```

**Partielle Anwendung / Currying**

```
def partial (f, *args):
    return lambda *moreargs: f(*(args + moreargs))

def revargs (f):
    return lambda *args: f(*(args[::-1]))

sqr = partial(revargs(pow), 2)

pow(5, 2) # 25
sqr(5)    # 25
```