Programming Concepts and Paradigms

Notizen (Herbstsemester 2018)

Patrick Bucher

16.01.2019

Inhaltsverzeichnis

1 Einführung

1.1 Programmiersprache

- Programmiersprache: formal konstruierte Sprache, entworfen um Befehle an Maschinen (speziell Computer) zu übermitteln; zum Schreiben von Programmen, welche
  - das Verhalten von Maschinen kontrollieren, oder
  - Algorithmen (schrittweises Verfahren zur Lösung eines Problems) beschreiben.
- Programme einer Sprache sind definiert durch:
  - Syntax: formale Struktur, Muster
  - Semantik: Bedeutung, Interpretation
- Turing-Vollständigkeit: universelle Programmierbarkeit, notwendige Bedingung für eine Programmiersprache
  - (turing-vollständige) Programmiersprachen sind prinzipiell gleich mächtig!
  - Kann ein Problem in *einer* turing-vollständigen Programmiersprache gelöst werden, kann es in *allen* turing-vollständigen Programmiersprachen gelöst werden.
  - Programmierparadigmen und -sprachen haben ihre Stärken und Schwächen.

1.2 Programmierparadigma

- Programmierparadigma: fundamentaler Programmierstil, eine bestimmte Art die Struktur und Elemente einer Software aufzubauen.
  - Paradigma: Denkweise, Lehrmeinung, Weltanschauung
- Programmierparadigmen:
  - imperativ: *Wie* ist ein Problem zu lösen? Schrittweise Anleitung in Befehlen; Angabe von *einzelnen Berechnungsschritten* (Lösungsweg)
    - strukturiert (Pascal): Sequenzen, Auswahl, Wiederholung (ohne `goto`)
    - prozedural (C): strukturiert mit Prozeduren/Funktionen/Unterprogrammen
      - Unterscheidung zwischen globalem und lokalem Gültigkeitsbereich
    - objektorientiert (Java): prozedural mit kommunizierenden Objekten
  - deklarativ: *Was* ist gegeben? Definition der Berechnungslogik ohne Angabe des Kontrollflusses. Beschreibung des Ergebnisses, nicht des Weges dorthin
    - funktional (Scheme): Definition von Funktionen, Rückgabewert nur von Parametern und nicht vom Zustand abhängig (referenzielle Transparenz)
    - logisch (Prolog): Herleitung der Lösung aufgrund gegebener Fakten und Regeln
- Programmiersprachen kombinieren meist ein Hauptparadigma mit anderen Paradigmen. Bei der imperativen Programmierung beinhalten "höhere" Paradigmen die Eigenschaften "tieferer" Paradigmen:
  - Assembler: imperativ
  - Pascal: strukturiert (und imperativ)
  - C: prozedural (und sturkturiert (und imperativ))
  - Java: objektorientiert (und prozedural (und strukturiert (und imperativ))) – mit funktionalen Elementen

2 Imperative Programmierung

- Grundidee der imperativen Programmierung: Programm als Sequenz von Anweisungen mit vorgegebenem Kontrollfluss
  - Verzweigungen: `if`/`else`, `switch`/`case`
  - Schleifen: `for`, `while`, `do`/`while`
  - Sprung: `goto`

2.1 Strukturierte Programmierung

- Strukturierte Programmierung: Imperative Programmierung ohne `goto`
  - Dijkstra: Go To Statement Considered Harmful, A Case against the GO TO Statement
  - Mit der hemmungslosen Verwendung von `goto` (*unbridled use of the go to statement*) wird es sehr schwierig nachzuvollziehen, wie weit ein Prozess bereits fortgeschritten ist.

2.2 Prozedurale Programmierung

- Prozedurale Programmierung: Strukturierte Programmierung mit Unterprogrammen/Funktionen/Routinen/Prozeduren – und lokalem Gültigkeitsbereich von Variablen
  - Auch Schleifen wären theoretisch überflüssig, da mit rekursiven Funktion zu bewerkstelligen. Sie sind aber syntaktisch äusserst praktisch und hatten sich damals schon eingebürgert.
  - Ohne lokalen Gültigkeitsbereich von Funktionsparametern wäre Rekursion nur sehr eingeschränkt möglich.

2.3 Zusammengesetzte Datentypen

Strukturen (C): fassen eine Liste von Variablen zusammen

```
struct point {
    int x;
    int y;
};

struct point p;
p.x = 1;
p.y = 2;

typedef struct {
    char[100] first_name;
    char[100] last_name;
    int year_of_birth;
} person;

person tom; // typedef: no `struct` needed
tom.first_name = "Thomas";
tom.last_name = "Miller";
tom.year_of_birth = 1964;
```

- Abstrakte Datentypen (ADT: Abstract Data Type): Semantik vorgegeben, Implementierung frei
  - Stack: Last In, First Out (LIFO)
    - `init`: leeren Stack initialisieren
    - `push`: Element auf Stack ablegen
    - `top`: letztes Element zurückgeben
    - `pop`: letztes Element vom Stack nehmen
    - `print`: alle Elemente ausgeben
    - Ein Stack kann in C mithilfe eines Arrays oder einer verketteten Liste implementiert werden. Die Schnittstelle wäre für beide Implementierung

gleich, die Laufzeiteigenschaften (Performance, Arbeitsspeicherbedarf) könnten Unterschiede aufweisen.
  - Queue: First In, First Out
  - Weitere: Matrix, Map, Ring Buffer
  - strukturierte/prozedurale Programmierung: Datenstrukturen und Funktionen voneinander getrennt (C: `struct` und Funktionen)
  - objektorientierte Programmierung: Datenstrukturen und Funktionen in einem Gefäss zusammengefasst (Java: Klasse mit Eigenschaften und Methoden)
    * Eigenschaften bilden den Zustand, Methoden das Verhalten ab
    * Sichtbarkeit: Einschränkung des Zugriffs (`public`, `protected`, default, `private`) auf Klassen- und Paketstufe
  - Java als Erweiterung von C um folgende Konzepte:
    * Klassen und Instanzen: `class`, `extends`, `super`, `instanceof`, `this`, `new`
    * Interfaces: `interface`, `implements`
    * Packages: `package`, `import`
    * Sichtbarkeit: `public`, `protected`, default, `private`
    * Exception Handling: `try`, `catch`, `finally`, `throw`, `throws`
  - Java-Referenzen als Einschränkung von C-Zeigern: Keine Zeigerarithmetik!

3 Prolog

- Prolog: deklarative, logische Programmiersprache
- Beschreibung des Resultats, nicht des Weges dorthin: *was*, nicht *wie*
- Wichtigste Mechanismen: Matching und automatisches Backtracking
- Funktionsweise von Prolog
  - Wissensdatenbank wird mit Fakten und Regeln gefüttert
  - Input: Es werden Anfragen (Queries) an die Wissensdatenbank gestellt
  - Output: Antworten werden mithilfe der Wissensdatenbank geliefert

3.1 Terme

Prolog-Terme:

1. Zahlen (numbers): Ganz- und Gleitkommazahlen
   - `324`, `-65`, `839.69`
2. Atome (atoms): mit Kleinbuchstaben beginnend oder mit Apostrophen umschlossen
   - `elephant`, `'Monty Python'`
3. Variablen (variables): mit Grossbuchstabe oder Underscore beginnend
   - `X`, `Elephant`, `_whatever`
   - anonyme Variable _: Platzhalter, falls Wert nicht von Interesse
4. Zusammengesetzte Terme (compound terms):
   - `is_neighbour('Node A', X)`

Term-Hierarchie:

- Einfache und zusammengesetzte Terme
  - Einfacher Term: Variable oder Atomarer Term
    - Atomarer Term: Atom oder Zahl
  - Zusammengesetzter Term: Kombination aus einfachen Termen
    - `smaller_of(A, B)`
    - `bigger_of(smaller_of(X, Y), bigger_of(A, B))`

Grundterme und Prädikate:

- Grundterme: Fakten
- Prädikate: Fakten oder Regeln
  - atomarer Term: Faktum
  - nicht-atomarer Term: Regel

Stelligkeit (Arity): Anzahl Argumente eines Prädikats für dessen Identifizierung.

- `bigger(A, B)`: `bigger/2`
  - zwei Argumente: Stelligkeit 2
- `division(dividend, divisor, result, remainder)`: `division/4`

3.2 Struktur eines Prolog-Programms

- Prolog-Programm: Liste von Klauseln
- Klausel: Fakten und Regeln (zusammengesetzte Terme)
- Prozedur: alle Klauseln zum gleichen Prädikat (Name und Stelligkeit gleich)

3.2.1 Fakten

Fakten: Prädikate gefolgt von einem Punkt; definieren etwas als bedingungslos wahr.

```
bigger(elephant, horse).
parent(peter, mary).
```

Prolog geht von einer geschlossenen Welt aus: Alles, was unbekannt ist, gilt als falsch!

3.2.2 Regeln

Regeln: Kopf (head) und Hauptteil (body), durch `:-` getrennt:

```
grandfather(X, Y) :-    % head
    father(X, Z),       % body, goal 1
    parent(Z, Y).       % body, goal 2
```

Das Komma verkettet zwei Ausdrücke als logisches Und (Konjunktion):

- Hornklauseln: `(p1 AND p2 AND ... AND pn) -> q`
- Prolog: `q :- p1, p2, ..., pn.`

3.2.3 Anfragen

Anfrage: Sequenz von 1..n Prädikaten gefolgt von einem Punkt:

```
?- bigger(elephant, dog).
true .
```

3.3 Funktionsweise

3.3.1 Matching

Matching:

- Zwei Terme sind identisch.
- Zwei Terme können durch Ersetzen von Variablen durch andere Terme identisch gemacht werden.

Matching wird per Gleichheits-Prädikat `=/2` abgefragt (Infix-Notation):

```
?- owns(alice, computer) = owns(X, computer).
X = alice.

?- owns(alice, computer) = owns(X, house).
false.
```

3.3.1.1 Atomare Terme

Atomare Terme matchen, wenn sie die gleiche Zahl oder das gleiche Atom sind:

```
?- =(tom, tom).
true.

?- tom = tom.
true.

?- 'Tom' = tom. % case sensitive!
false.

?- 'tom' = tom
true.
```

3.3.1.2 Variable und Term

Ist einer der Terme eine Variable, matchen die Terme, und die Variable erhält den Wert des anderen Terms:

```
?- alice = X.
X = alice.

?- owns(house) = X.
X = owns(house).

?- X = Y.
X = Y.

?- X = Y, X = alice.
X = Y, Y = alice.

?- X = Y, X = alice, Y = bob.
false.
```

3.3.1.3 Zusammengesetzte Terme

Zusammengesetzte Terme stimmen überein, wenn:

1. Sie vom gleichen Funktor mit der gleichen Stelligkeit sind.
2. Alle korrespondierenden Argumente übereinstimmen.

```
?- meet(drink(alice), eat(X)) = meet(Y, eat(bob)).
X = bob
Y = drink(alice).

?- meet(X, X) = meet(drink(alice), bob).
false.

?- meet(alice, bob) = meet(Y).
false.
```

3.3.2 Backtracking

Eine Anfrage führt zu einer Beweissuche im Suchbaum. Die Abarbeitung erfolgt:

- in der Wissensdatenbank von oben nach unten
- im Ausdruck von links nach rechts

Kann ein Ausdruck in einem Suchbaum nicht aufgelöst werden, kehrt die Abarbeitung automatisch auf die letzte Verzweigung zurück. Für den nicht aufgelösten Ausdruck wird nun die nächste Variante probiert. Sind keine weiteren Varianten verfügbar, wird die Abarbeitung abgebrochen.

Beispiel:

```
f(a).
f(b).
g(a).
g(b).
h(b).

k(X) :- f(X), g(X), h(X). % rule1

?- k(Y).
Y = b
```

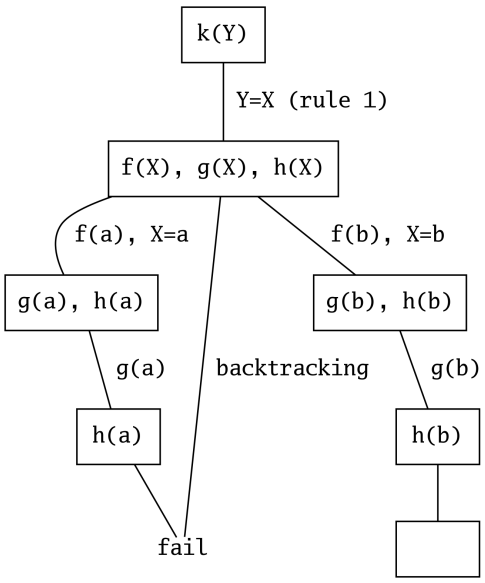


Abbildung 1: Suchbaum (Tiefensuche)

Die Reihenfolge von Regeln und Zielen ist wichtig!

1. Die Reihenfolge der Regeln und Ziele wirkt sich auf die Performance aus.
   - Triviale Fälle (einfache Regeln, Rekursionsbasis) sollten immer oberhalb von nicht-trivialen Fällen (allgemeine Regel, Rekursionsanweisung) stehen.
2. Eine Tiefensuche kann aufgrund einer nicht-terminierenden Rekursion unendlich laufen.
   - Bei zusammengesetzten Prädikaten sollten die einfachen Ziele vor den rekursiven Ziele stehen.

3.4 Beispiele

Einträge in die Wissensdatenbank:

```
bigger(elephant, horse).
bigger(horse, dog).
bigger(horse, sheep).
```

Anfrage:

```
?- bigger(dog, elephant).
false.

?- bigger(elephant, horse).
true.

?- bigger(horse, X).
X = dog ;
X = sheep.

?- bigger(elephant, dog).
false
% elephant > horse und horse > dog,
% aber bigger ist nicht transitiv definiert!
```

3.4.1 Transitive Hülle

```
bigger(X, Y) :- bigger(X, Z), bigger(Z, Y).
```

Anfrage:

```
?- bigger(elephant, dog).
true

?- bigger(X, dog).
X = horse ;
X = elephant ;
false.

?- bigger(elephant, X).
X = horse ;
X = dog ;
X = sheep ;
false.

?- bigger(elephant, X), bigger(X, dog).
X = horse ;
false.
```

```
?- bigger(horse, X), bigger(X, dog).
false.
```

3.4.2 Matching

Zwei Punkte sind durch eine vertikale (bzw. horizontale) Linie verbunden, wenn sie die gleiche X-Koordinate (bzw. Y-Koordinate) haben:

```
vertical(line(point(X, _), point(X, _))).
horizontal(line(point(_, Y), point(_, Y))).
```

Anfrage:

```
?- vertical(line(point(1, 1), point(1, 5))).
true.

?- horizontal(line(point(1, 2), point(3, X))).
X = 2.

?- horizontal(line(point(1, 2), P)).
P = point(_G4232, 2).
```

3.5 Arithmetik und Operatoren

Der Gleichheitsoperator `=/2` macht Matching:

```
?- X = 1 + 2.
X = 1+2.
```

Für die Zuweisung wird der `is/2`-Operator verwendet:

```
?- X is 1 + 2.
X = 3.
```

Die Dokumentation gibt den Operator als `-Number is +Expr` an.

- `-`: der Operand muss ungebunden sein
- `+`: der Operand muss gebunden sein
- `?`: der Operand kann gebunden oder ungebunden sein

Nur gebundene Operanden können an ungebundene Operanden zugewiesen werden.

Folgende arithmetische Operatoren sind vordefiniert:

- `+`: Addition
- `-`: Subtraktion
- `/`: Division
- `**`: Potenz
- `//`: Ganzzahldivision

- `mod`: Modulo (Rest der Ganzzahldivision)
- `sqrt`: Quadratwurzel

Beispiel:

```
?- X is 3 + 2.
X = 5.

?- X is 3 - 2.
X = 1.

?- X is 2 * 4.
X = 8.

?- X is 10 / 3.
X = 3.3333333333333335.

?- X is 2 ** 3.
X = 8.

?- X is 10 // 3.
X = 3.

?- X is 10 mod 3.
X = 1.

?- X is sqrt(16).
X = 4.
```

Es stehen folgende Vergleichsoperatoren zur Verfügung:

- >: grösser als
- <: keiner als
- >=: grösser-gleich
- =<: kleiner-gleich
- =:=: Gleichheit
- =\=: Ungleichheit
- ==: Identität

Beispiel:

```
?- 4 > 5.
false.

?- 4 < 5.
true.
```

```
?- 4 >= 5.
false.

?- 4 =< 5.
true.

?- 4 =:= 5.
false.

?- 4 =\= 5.
true.

?- 4 == 5.
true.
```

Operatoren sind Prädikate, die mit Infix-Notation verwendet werden können:

```
?- 11 > 7.
true.

?- >(11, 7).
true.
```

3.5.1 Eigene Operatoren definieren

Mit dem Direktiven `op/3` können eigene Operatoren definiert werden:

- Syntax: `:- op(+Precedence, +Type, :Name)`
  - `:-` steht für eine Direktive
  - Precedence: Ganzzahl zwischen 1 und 1200 (je tiefer, desto stärker bindend)
  - Type: Infix, Präfix, Postfix
    - Infix (Operator zwischen Operanden): xfx, xfy, yfx, z.B. + (Addition)
    - Präfix (Operator vor Operand): fx, fy, z.B. + (Vorzeichen)
    - Postfix (Operator nach Operand): xf, yf
    - x: Operand mit Präzedenz kleiner derjenigen von Operator f
    - y: Operand mit Präzedenz kleiner/gleich derjenigen von Operator f
    - yfx: linsassoziativ
    - xfy: rechtsassoziativ
  - Name: Prädikatsname

Beispiel:

```
bigger(elephant, dog).

?- op(1150, xfx, bigger).
true.
```

```
?- elephant bigger dog.
true.
```

3.5.2 Verknüpfung von Zielen

- Konjunktion: UND-Logik mit Komma-Operator `,` (Prädikat `,/2`)
- Disjunktion: ODER-Logik mit Semikolon-Operator `;` (Prädikat `;/2`)

Disjunktionen können auch implizit als Liste von Regeln ausgedrückt werden:

```
% loud OR smelly OR stupid
annoying(X) :- loud(X); smelly(X); stupid(X).

annoying(X) :- loud(X).   % loud
annoying(X) :- smelly(X). % OR smelly
annoying(X) :- stupid(X). % OR stupid
```

3.6 Rekursion

Beispiel (`bigger/2`):

```
bigger(X, Y) :- bigger(X, Y).               % simple case (Rekursionsbasis)
bigger(X, Y) :- bigger(X, Z), bigger(Z, Y). % general case (Rekursionsanweisung)
```

Fakultätsberechnung (`fak/2`):

```
fak(0, 1).
fak(A, B) :-
    A > 0,
    A1 is A - 1,
    fak(A1, B1),
    B is A * B1.

?- fak(5, X). % 5! = 120
X = 120
false.

?- fak(4, 24). % is 4! = 24? yes
true.
```

Fibonacci-Zahlen (`fib/2`):

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
```

```
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.

?- fib(7, X). % seventh fibonacci number? 13
X = 13.

?- fib(6, 8). % sixth fibonacci number is 8? yes
true.
```

3.6.1 Optimierung mittels Endrekursion

- Problem: Rekursion ist oft ineffizient
  - der Stack ist mit laufenden Prozeduren gefüllt
  - manche Zwischenergebnisse müssen mehrmals berechnet werden (Fibonacci)
- Endrekursion (tail recursion) gilt für eine Prozedur, wenn:
  - diese nur einen rekursiven Aufruf hat
  - dieser der letzte Aufruf der letzten Klausel der Prozedur ist
  - alle anderen Aufrufe deterministisch sind
- Endrekursion kann als Iteration ausgeführt werden
  - kein Backtracking notwendig
  - kein zusätzlicher Speicherbedarf
  - Umwandlung rekursive in endrekursive Prozedur nicht trivial
  - benötigt zusätzliche Akkumulator-Argumente

Beispiel (Fibonacci-Zahlen mittels Endrekursion):

```
fib(N, F) :- fib(N, 0, 1, F). % accumulator
fib(0, A, _, A).              % simple case
fib(N, A, B, F) :-            % general case
    N1 is N - 1,
    N1 >= 0,                  % avoid underflow
    Sum is A + B,             % accumulator Sum
    fib(N1, B, Sum, F).       % tail-recursive call
```

C-Implementierung zum Vergleich:

```
int fib(int n) {
    return tailrec_fib(n-1, 0, 1);
}

int tailrec_fib(int n, int a, int b) {
    if (n >= 0) {
```

```
        return tailrec_fib(n-1, b, a+b);
    }
    return a;
}
```

3.6.2 Optimierung mittels Assertions (Memoization)

Fakten und Regeln können zur Laufzeit hinzugefügt und gelöscht werden. Dadurch lassen sich Programme durch Caching optimieren (Memoization).

Beispiel (`bigger.pl`):

```
:- dynamic bigger/2. % Prädikat modifizierbar machen.
bigger(elephant, horse).
bigger(horse, dog).
bigger(horse, sheep).
```

`listing/1` zeigt die Fakten und Regeln zu einem Prädikat an:

```
?- listing(bigger).
:- dynamic bigger/2.
bigger(elephant, horse).
bigger(horse, dog).
bigger(horse, sheep).
```

`asserta/1` fügt eine neue Regel *zuoberst,* `assertz/1` *zuunterst* ein:

```
?- asserta(bigger(horse, hamster)).
?- assertz(bigger(dog, hamster)).
?- listing(bigger).
bigger(horse, hamster).
bigger(elephant, horse).
bigger(horse, dog).
bigger(horse, sheep).
bigger(dog, hamster).
```

`retract/1` entfernt eine Regel:

```
?- retract(bigger(horse, dog)).
?- retract(bigger(horse, sheep)).
?- listing(bigger).
bigger(horse, hamster).
bigger(elephant, horse).
bigger(dog, hamster).
```

3.6.2.1 Anwendungsbeispiel: Optimierung Fibonacci-Zahlen

Bei der rekursiven Berechnung von Fibonacci-Zahlen werden manche Zwischenresultate mehrmals berechnet:

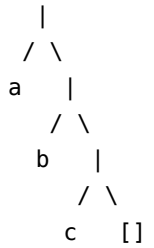
- Beispiel: `f(4)=f(3)+f(2)`, `f(3)=f(2)+f(1)` (`f(2)` doppelt)
- Bei `f(5)` wird `f(2)` fünf mal berechnet.
- Wird das Zwischenergebnis von `f(2)` als neue Regel abgespeichert, müsste es nur einmal berechnet werden.

```
:- dynamic fib/2.
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N-1,
    N2 is N-2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1+F2,
    asserta(fib(N, F)). % memoize result as a new rule
```

3.7 Listen

- Liste: endliche Sequenz von Elementen
- Notation: in eckigen Klammern, Elemente durch Kommas getrennt
- leere Liste: `[]`
- Elemente: beliebige Prolog-Terme (Atome, Zahlen, Listen)
- Aufbau: Rekursiv, aus zwei Teilen bestehend
  1. Listenkopf (head): erstes Element
  2. Listenschwanz (tail): Liste übriger Elemente

Liste `[a, b, c]` als Baumdarstellung (terminiert durch leere Liste als Rekursionsbasis):



Der Listen-Operator `|` (Pipe) teilt eine Liste in die Variablen Head und Tail auf:

```
?- [a, b, c] = [Head | Tail].
Head = a
Tail = [b, c].
```

Bestimmte Elemente aus der Liste lesen (Beispiel: zweites und drittes):

```
?- [_, Second, Third | _] = [a, b, c, d, e, f].
Second = b.
Third = c.
```

3.7.1 Listenoperationen

3.7.1.1 Listenzugehörigkeit

Ist ein Element X in einer Liste L enthalten?

- Relation `mem(X, L)`:
  - `mem(a, [a, b, c])` trifft zu
  - `mem(d, [a, b, c])` trifft nicht zu
  - `mem(b, [a, [b, c]])` trifft nicht zu
  - `mem([b, c], [a, [b, c]])` trifft zu
- Element X ist in der Liste L enthalten, wenn:
  - X der Kopf von L ist
  - X im Schwanz von L vorkommt

```
mem(X, [X | _]).      % X as list header (ignore tail)
mem(X, [_ | Tail]) :- % X in list tail (ignore header)
    mem(X, Tail).

?- mem(a, [a, b, c]).
true .

?- mem(d, [a, b, c]).
false.

?- mem(b, [a, [b, c]]).
false.

?- mem([b, c], [a, [b, c]]).
true .
```

`mem` kann zur Auflistung aller Listenelemente oder zum Finden von Listen mit speziellen Eigenschaften verwendet werden:

```
?- mem(X, [a, b, c]). % list all list members
X = a ;
X = b ;
X = c ;
false.

?- mem(foo, L). % create a list containing foo
L = [foo|_2502] ;
```

```
L = [_2500, foo|_2508] .

?- mem(a, L), mem(b, L), mem(c, L). % find list containing a, b and c
L = [a, b, c|_2700] ;
L = [a, b, _2689, c|2706] .

?- L = [_, _, _], mem(a, L), mem(b, L), mem(c, L). % permutations of L
L = [a, b, c] ;
L = [a, c, b] ;
L = [b, a, c] ;
L = [b, c, a] ;
L = [c, a, b] ;
L = [c, b, a] ;
false.
```

Prolog stellt bereits das Prädikat `member/2` mit der beschriebenen Semantik zur Verfügung.

3.7.1.2 Listenkonkatenation

Ergeben die Listen L1 und L2 zusammengehängt die Liste L3?

- Relation `conc(L1, L2, L3)`
  - `conc([a, b], [c, d], [a, b, c, d])` trifft zu
  - `conc([a, b], [c], [a, b, c, d])` trifft nicht zu
  - `conc([a, b, c], [d], [a, b, c, d])` trifft zu
  - `conc([a, b, c], [], [c, b, a])` trifft nicht zu
- L1 und L2 ergeben zusammengehängt L3, wenn:
  1. L1 leer ist, und L2 und L3 gleich sind.
  2. L1 nicht leer ist; Kopf und Schwanz hat: `[X | L1]`
     - `[X | L3]` als Resultat der Konkatenation,
     - wobei L3 die Konkatenation aus L1 und L2 ist

```
conc([], L, L).                 % case 1
conc([X | L1], L2, [X | L3]) :- % case 2
    conc(L1, L2, L3).
```

Der zweite Fall tritt solange ein, bis die Abspaltung des Listenkopfes X von L1 eine leere Liste ergibt, wobei der erste Fall eintritt.

Anwendung (Verkettung, Subtraktion, Zerlegung in alle möglichen Teillisten):

```
?- conc([a, b], [c, d], L).
L = [a, b, c, d].

?- conc(X, [b, c], [a, b, c, d]).
X = [a, b] .
```

```
?- conc(L1, L2, [a, b, c]).
L1 = [],
L2 = [a, b, c] ;
L1 = [a],
L2 = [b, c] ;
L1 = [a, b],
L2 = [c] ;
L1 = [a, b, c],
L2 = [];
```

3.8 Cut `!/0`

Cut bindet die bisherige Wahl innerhalb der aktuellen Klausel, sodass der vor dem Cut-Operator eingeschlagene Pfad mittels Backtracking nicht mehr verlassen werden kann.

- Green Cut: verändert die Semantik eines Prädikats nicht
  - zur Performanceoptimierung (durch bereits erfüllte Regeln redundant gewordene Regeln nicht mehr prüfen)
- Red Cut: verändert/bestimmt die Semantik eines Prädikats
  - Anwendungsbeispiel: Lösungsfindung nach gefundener Lösung abbrechen
  - Verwendung führt zu nicht rein deklarativen, teilweise schwer verständlichen Programmen

3.8.1 Beispiel zur Funktionsweise

Beispiel ohne Verwendung von `!/0` (`cut1.pl`):

```
p(X) :- a(X).       % Rule r1
p(X) :- b(X), c(X). % Rule r2
p(X) :- d(X).       % Rule r3

a(1).
b(2). b(3).
c(2). c(3).
d(4).

?- p(X).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4
```

Suchbaum:

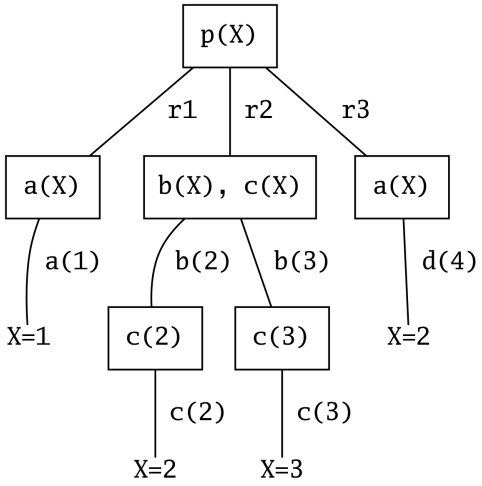


Abbildung 2: Suchbaum zu `cut1.pl`

Beispiel unter Verwendung von `!/0` (`cut2.pl`):

```
p(X) :- a(X).          % Rule r1
p(X) :- b(X), !, c(X). % Rule r2 (modified)
p(X) :- d(X).          % Rule r3

a(1).
b(2). b(3).
c(2). c(3).
d(4).

?- p(X).
X = 1 ;
X = 2
```

3.8.2 Anwendungsbeispiel: Maximum berechnen

Relation `max/3` zum Finden der grösseren der beiden Zahlen:

```
max(X, Y, X) :- X >= Y. % r1: X grösser/gleich Y
max(X, Y, Y) :- X < Y.  % r2: X kleiner als Y
```

Die beiden Regeln `r1` und `r2` schliessen sich gegenseitig aus. Optimierungsversuch mit `!/0` (*green cut*):

```
max(X, Y, X) :- X >= Y, !. % r1: X grösser/gleich Y
max(_X, Y, Y).             % r2: sonst muss Y grösser sein
```

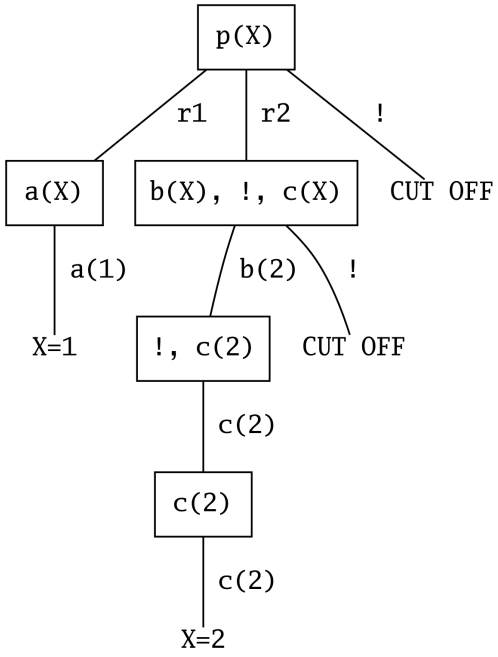


Abbildung 3: Suchbaum zu `cut2.pl`

```
?- max(77, 55, 55).
true. % müsste false sein!
```

Umformulierte Version, die auch mit drei instanzierten Variablen funktioniert:

```
max(X, Y, Z) :- X >= Y, !, X = Z.
max(_X, Y, Y).
```

3.8.2.1 Anwendungsbeispiel: Listenzugehörigkeit

Relation `mem/2` zum Prüfen, ob X ein Element der Liste L ist:

```
mem(X, [X | _]).
mem(X, [_ | Tail]) :- mem(X, Tail).

?- mem(a, [a, b, c, a, b, c, a, b, c]).
true ;
true ;
true ;
false.
```

Es werden alle Lösungen gefunden (`a` ist dreimal in der Liste enthalten).

Soll nur eine Lösung gefunden werden, muss nur der erste Fakt als Regel mit `!/0` umformu-

liert werden (*red cut*):

```
mem(X, [X | _]) :- !. % cut after first match
mem(X, [_ | Tail]) :- mem(X, Tail).

?- mem(X, [a, b, c, a, b, c, a, b, c]).
X = a.
```

3.9 Negation

Gegeben sei diese Wissensdatenbank:

```
food(hamburger).
food(chickennuggets).
food(fishnchips).
food(salad).
vegan(salad).
```

Um auszudrücken, dass Tom alles ausser veganes Essen mag, könnte man für jedes Essen ein Prädikat definieren:

```
likes(tom, hamburger).
likes(tom, chickennuggets).
likes(tom, fishnchips).
```

Dazu müsste aber die Wissensdatenbank bei jedem neuen Essen ergänzt werden. Einfacher lässt sich der Sachverhalt mittels `fail`-Prädikat als Negation ausdrücken:

```
likes(tom, X) :- % special negation rule
    vegan(X),    % is X vegan
    !,           % then cut
    fail.        % and evaluate to false
likes(tom, X) :- % general positive rule
    food(X).     % evaluate everything else to true
```

3.9.1 Verschiedenheit

Verschiedenheit zweier Ausdrücke X und Y kann unterschiedlich interpretiert werden:

1. X und Y sind nicht buchstäblich gleich.
2. X und Y matchen nicht.
3. X und Y haben unterschiedliche arithmetische Werte.

Die Fälle 1 und 3 werden durch Standardoperatoren abgedeckt. Fall 2 lässt sich folgendermassen ausdrücken:

```
different(X, X) :- % two matching variables are not different
    !,
```

```
    fail.
different(_X, _Y). % two non-matching variables are different

-? different(X, tom).
false.

-? different(3, tom).
true.

-? different(1 + 2, 3).
true.

-? different(mary, tom).
true.
```

3.9.2 NOT-Prädikat

Allgemein lässt sich Negation in Prolog folgendermassen ausdrücken:

```
not(P) :-
    P,     % if true
    !,     % cut
    fail.  % and fail
not(_P) :-
    true.
```

Prolog hat ein eingebautes Prädikat `not/1` mit der alternativen Syntax `\+`:

```
?- not(3 = 2).
true.

?- \+ 3 = 2.
true.

?- not(3 = 3).
false.

?- \+ 3 = 3.
false.
```

3.9.3 Negation und *Closed World Assumption*

Alles, was nicht in der Wissensdatenbank abgelegt ist, wird von Prolog als falsch interpretiert:

```
round(ball).

?- round(ball).       % known fact: true
true.

?- round(earth).      % unknown: false
false.

?- not(round(earth)). % unknown (false) negated: true
true.
```

Dies kann zu unerwarteten Effekten führen:

```
works_on_linux(idle).
works_on_linux(pycharm).
requires_java(pycharm).

bloatfree(IDE) :-
    not(requires_java(IDE)).

?- works_on_linux(X), bloatfree(X). % as expected
X = idle.

?- bloatfree(X), works_on_linux(X). % something went wrong
false.
```

- Im ersten Aufruf matcht `works_on_linux(X)` zunächst mit `X = idle`.
  - `requires_java(idle)` ergibt keinen Treffer, es evaluiert zu `false`.
  - Das Ergebnis `false` wird negiert und ergibt `true`, sodass `idle` ein Treffer ist.
  - Das Prädikat `works_on_linux(idle)` matcht ebenfalls.
  - Das Gesamtprädikat evaluiert zu `true`.
- Im zweiten Aufruf matcht `requires_java(IDE)` nur auf `IDE = pycharm`.
  - Das Ergebnis wird mit `not` negiert, sodass `bloatfree` zu `false` evaluiert.
  - Das zweite Prädikat wird aufgrund der AND-Logik nicht mehr evaluiert.
  - Das gesamte Prädikat evaluiert zu `false`.

Ein Prädikat `not(predicate(X))` bedeutet:

> existiert *kein* X, wofür `predicate` gilt? (universelle Quantifikation)

und nicht:

> existiert ein X, wofür `predicate` *nicht* gilt? (existenzielle Quantifikation)

Merke: Negation sollte nur mit instanziierten Variablen verwendet werden!

3.10 Constraint Logic Programming (CLP)

- *Constraint Satiscaftion Problems* (CSP) sind definiert durch:
  1. Variablen
  2. mit Wertebereichen (Domains),
  3. die bestimmte Bedingungen (Constraints) erfüllen müssen
- Es wird eine Wertezuweisung zu den Variablen gesucht, sodass alle Bedingungen erfüllt sind.
- Es gibt drei CSP-Domänen:
  1. CLP-R: für reelle Zahlen (Library `clpr`)
  2. CLP-Q: für rationale Zahlen (Library `clpq`)
  3. CLP-FD: für finite Domänen (eigene Wertebereiche, Library `clpfd`)

3.10.1 CLP-R

Erfordert `clpr` libary:

```
:- use_module(library(clpr)).
```

Beispiel (Celsius/Fahrenheit-Umrechung):

```
:- use_module(library(clpr)).

% "old" solution
convert(Celsius, Fahrenheit) :-
    Celsius is (Fahrenheit - 32) * 5 / 9.

% CLP-R solution
convert_clpr(Celsius, Fahrenheit) :-
    { Celsius = (Fahrenheit - 32) * 5 / 9 }.

?- convert(C, 95).
C = 35.

?- convert(35, F).              % only works in one direction
ERROR

?- convert_clpr(C, 95).
C = 35.0 .

?- convert_clpr(35, F).         % works in both directions
F = 95.0 .

?- convert_clpr(C, F).
{F=32.0+1.7999999999999998*C}. % even works without arguments, displays the rule
```

Beispiel (Fibonacci-Zahlen):

```
:- use_module(library(clpr)).
fib_clpr(N, F) :- { N = 0, F = 0 }.
fib_clpr(N, F) :- { N = 1, F = 1 }.
fib_clpr(N, F) :-
    { N >= 2, F = F1 + F2, N1 = N - 1, N2 = N - 2 },
    fib_clpr(N1, F1),
    fib_clpr(N2, F2).

?- fib_clpr(5, F).  % forward lookup
F = 5.0 .

?- fib_clpr(N, 55). % reverse lookup
N = 10.0 .
```

Beispiel (Zahlenrätsel):

> Ludwig ist 16 Jahre alt. Damit ist Ludwig genau doppelt so alt, wie Luise war, als Ludwig so alt war, wie es Luise jetzt ist! Wie alt ist Luise?

```
:- use_module(library(clpr)).
?- { LudwigNow = 16,               % Ludwig ist 16 Jahre alt.
     LudwigNow = 2 * LuiseThen,    % Damit ist Ludwig genau doppelt so alt wie Luise war,
     LudwigThen = LuiseNow,        % als Ludwig so alt war, wie es Luise jetzt ist!
     LuiseNow = LuiseThen + N,     % (Luise und Ludwig sind beide...
     LudwigNow = LudwigThen + N }. %  ... um N Jahre gealtert.)
LudwigNow = 16.0,
LuiseThen = 8.0,
LudwigThen = LuiseNow, LuiseNow = 12.0,
N = 4.0 .
```

3.10.2 CLP-FD

Erfordert `clpfd` libary:

```
:- use_module(library(clpfd)).
```

Wichtige Prädikate:

- `in/2`: Wertebereich einer Variable festlegen
- `ins/2`: Wertebereich für eine Liste von Variablen festlegen
- `all_distinct/1`: Sicherstellen, dass alle Variablen in einer Liste unterschiedliche Werte haben
- `label/1`: allen Variablen in der Liste Werte zuweisen (konkrete Lösungen finden)
- alle FD-Vergleichsoperatoren benötigen das Präfix #

Beispiel (interaktive Session):

```
?- use_module(library(clpfd)).
true.

?- X in 1..3, X #> 2.
X = 3.

?- [X, Y] ins 1..2, X #< Y.
X = 1.
Y = 2.

?- [X, Y] ins 1..2, all_distinct([X, Y]).
X in 1..2,
all_distinct([X, Y]),
Y in 1..2.

?- [X, Y] ins 1..2, all_distinct([X, Y]), label([X, Y]). % find specific solutions
X = 1,
Y = 2 ;
X = 2,
Y = 1.
```

3.10.2.1 Anwendungsbeispiel: Sudoku-Solver

Problemstellung:

- ein 9x9-Grid aufgeteilt in 9 Blöcke von 3x3 Feldern
- in jedem Feld muss eine Zahl zwischen 1 und 9 (inklusiv) stehen
- pro Zeile, Spalte und Block darf jede Ziffer nur einmal auftauchen

Lösung:

```
:- use_module(library(clpfd)).

sudoku(Rows) :-
    append(Rows, Vs),                  % join all Rows to one flat list
    Vs ins 1..9,                       % define domain of all list items
    maplist(all_distinct, Rows),       % unique values per row
    transpose(Rows, Columns),          % pivot row/col grid to col/row grid
    maplist(all_distinct, Columns),    % unique values per column
    Rows = [A, B, C, D, E, F, G, H, I], % extract rows to check blocks
    blocks(A, B, C),                   % check 3 blocks of rows A, B, C
    blocks(D, E, F),                   % check 3 blocks of rows D, E, F
    blocks(G, H, I),                   % check 3 blocks of rows G, H, I
```

```
    maplist(label, Rows).               % map solution to variables

blocks([], [], []).                     % basic case: empty lists
% the three first elements of three adjacent form a block
blocks([A, B, C | T1], [D, E, F | T2], [G, H, I | T3]) :-
    all_distinct([A, B, C, D, E, F, G, H, I]),
    blocks(T1, T2, T3).
```

Aufruf:

```
Puzzle = [                          % define the grid
    [5, 3, _, _, 7, _, _, _, _],
    [6, _, _, 1, 9, 5, _, _, _],
    [_, 9, 8, _, _, _, _, 6, _],
    [8, _, _, _, 6, _, _, _, 3],
    [4, _, _, 8, _, 3, _, _, 1],
    [7, _, _, _, 2, _, _, _, 6],
    [_, 6, _, _, _, _, 2, 8, _],
    [_, _, _, 4, 1, 9, _, _, _],
    [_, _, _, _, 8, _, _, 7, 9]
],
Puzzle = [A, B, C, D, E, F, G, H, I], % map grid to rows
sudoku([A, B, C, D, E, F, G, H, I]).  % call solver for list of rows
```

Lösung:

```
A = [5, 3, 4, 6, 7, 8, 9, 1, 2],
B = [6, 7, 2, 1, 9, 5, 3, 4, 8],
C = [1, 9, 8, 3, 4, 2, 5, 6, 7],
D = [8, 5, 9, 7, 6, 1, 4, 2, 3],
E = [4, 2, 6, 8, 5, 3, 7, 9, 1],
F = [7, 1, 3, 9, 2, 4, 8, 5, 6],
G = [9, 6, 1, 5, 3, 7, 2, 8, 4],
H = [2, 8, 7, 4, 1, 9, 6, 3, 5],
I = [3, 4, 5, 2, 8, 6, 1, 7, 9].
```

3.11 Weitere vordefinierte Prädikate

- `trace/0`: Abarbeitung mitverfolgen
- `guitrace/0`: Abarbeitung in GUI mitverfolgen

4 Racket (Scheme)

- Racket (Scheme): deklarative, funktionale Programmiersprache

- homoikonisch: selbstrepräsentierbar (Programme sind Datenstrukturen derselben Sprache)
- Auf LISP basierend: List Processing (jedes Programm ist eine Liste)
- Funktionen stehen im Zentrum
  - keine Seiteneffekte (referenzielle Transparenz)
  - First-Class Funktionen: Funktionen als gleichberechtigte Datenobjekte
  - Lösen von Aufgaben mittels Dekomposition des Problems und Komposition der Teillösungen zu einer Gesamtlösung

4.1 Grundlagen

4.1.1 Form

```
(<operator> <operand 1> <operand 2> ... <operand n>)
```

- in runde Klammern eingeschlossen
- Präfix-Schreibweise: zuerst Operator (bzw. Funktion), dann Operanden
- nicht festgelegte Reihenfolge
- beliebig verschachtelbar
- paarweise Klammern

4.1.2 Auswertungsregeln

Ausdrücke geben folgende Werte zurück:

- selbstauswertender Ausdruck: Wert der Zahl/des boolschen Ausdrucks
- Name: Wert, der in der jeweiligen Umgebung mit diesem Namen assoziiert wird
- eingebauter Operator: Sequenz der Instruktionen, welche entsprechende Operation durchführen
- Kombination:
  - Unterausdrücke in beliebiger Reihenfolge
  - Funktion des linksstehenden Unterausdrucks (Operator) auf Operanden anwenden
- Form mit Schlüsselwort: Funktion auf Operanden anwenden

4.1.3 Definition von Namen

```
(define <identifier> <expression>)
(define PI 3.1459)
```

- `identifier` ist ein beliebiger Name, der auch Sonderzeichen enthalten kann
  - Unterscheidung: case-sensitive
- `expression` kann ein beliebiger Ausdruck sein (Konstante, Variable, Funktionsaufruf)

4.1.4 Definition von Funktionen

```
(define (<identifier> <formal parameters>)
    <expression>)
(define (power-of-two (x))
    (* x x))
```

4.1.5 Auswertung von Funktionen

- strikte Auswertung: alle Argumente werden vor dem Funktionsaufruf ausgewertet
- Bedarfsauswertung (*lazy evalutation*): Argumente werden unausgewertet übergeben; Auswertung erfolgt erst, wenn die Werte benötigt werden
  - potenziell zeitsparend, da Auswertungen vermieden werden können
  - Kurzschlussauswertung: Auswertung logischer Ausdrücke kann vorzeitig abgebrochen werden

4.2 Atomare Ausdrücke

- Selbstauswertende Ausdrücke: Bezeichner und Bezeichnetes sind das Gleiche
- Zahlen (selbstauswertend)
  - Ganzzahlen (`integer`): 23
  - Rationale Zahlen (`rational`): ½, ¼ oder 0.5, 0.25
  - Reelle Zahlen (`real`): ±∞
  - Irrationale Zahlen: `pi` = `#3.14159`
  - Komplexe Zahlen (`complex`): √(-2)
- Boolsche Werte: `true` und `false` (selbstauswertend)
  - `#t` → `true`
  - `#f` → `false`
- Operatoren für ganze Zahlen:
  - `+`: Addition
    - `(+ 3 -2 5)` → `6`
  - `-`: Subtraktion
    - `(- 31 17 7 5)` → `2`
  - `*`: Multiplikation
    - `(* 2 3)` → `6`
  - `/`: Division
    - `(/ 128 4 2 2 4)` → `2`
  - `quotient`: Ganzzahl-Division
    - `(quotient 10 3)` → 3
  - `remainder`: Divisionsrest
    - `(modulo -10 3)` → `1`
  - `expt`: Potenz
    - `(expt 2 10)` → `1024`
  - `modulo`: Modulo

        * `(modulo -10 3)` → `2`
    - `abs`: Absolutwert
        * `(abs -3)` → `3`
    - `max`: Maximum
        * `(max 3 6 1 9 6)` → `9`
    - `min`: Minimum
        * `(min 3 6 1 9 6)` → `1`
    - `lcm`: kgV (kleinstes gemeinsames Vielfaches)
        * `(lcm 30 4)` → `60`
    - `gcd`: ggT (grösster gemeinsamer Teiler)
        * `(gcd 1024 768)` → `256`
    - `numerator`: Zähler des einfachst möglichen Bruchs
        * `(numerator 7.25)` → `4`
    - `denominator`: Nenner des einfachst möglichen Bruchs
        * `(denominator 7.25)` → `29.0`
- Operatoren für reelle und komplexe Zahlen:
    - `exp`: Exponentialfunktion (Potenz zur Basis `e`)
        * `(exp 1)` → `2.718281828459045`
    - `sin`: Sinus
        * `(sin 1.5)` → `0.9974949866040544`
    - `cos`: Cosinus
        * `(cos 1.5)` → `0.0707372016677029`
    - `tan`: Tangens
        * `(tan 1)` → `1.5574077246549023`
    - `expt`: Potenz
        * `(expt 1.5 2.5)` → `2.7556759606310752`
    - `log`: Logarithmus zur Basis `e`
        * `(log (exp 1))` → `0`
    - `sqr`: Quadrat
        * `(sqr 4.1)` → `16.81`
    - `sqrt`: Quadratwurzel
        * `(sqrt 17.3)` → `4.159326868617084`

4.3 Bedingte Ausdrücke

4.3.1 Vergleichsoperatoren

- =: gleich
    - `(= 5 5)` → `#t`
    - `(= 3 4)` → `#f`
- >: grösser als
    - `(> 7 2)` → `#t`
    - `(> 2 7)` → `#f`
    - `(> 9 7 5 3 1)` → `#t`

    - `(> 9 7 5 3 3)` → `#f`
- `<`: keliner als
    - `(< 2 5)` → `#t`
    - `(< 5 2)` → `#f`
- `>=`: grösser/gleich
    - `(>= 5 5)` → `#t`
    - `(>= 3 4)` → `#f`
    - `(>= 9 7 5 3 3)` → `#t`
- `<=`: kleiner/gleich
    - `(<= 5 5)` → `#t`
    - `(<= 5 3)` → `#f`

4.3.2 Logische Verknüpfungen

- Sequenzielle Auswertung mit Kurzschlussverfahren
- Fehler, wenn ein Ausdruck weder `true` noch `false` ergibt
- Auswertung: von links nach rechts
- `and`: UND-Logik
    - Kurzschluss: `false`, bei erstem Teilausdruck, der zu `false` evaluiert
    - `(and (> 5 3) (< 1 9))` → `#t`
    - `(and (> 5 3) (= 1 9))` → `#f`
    - `(and (> 5 3) (+ 1 9))` → `Error`
- `or`: OR-Logik
    - Kurzschluss: `true`, bei erstem Teilausdruck, der zu `true` evaluiert
    - `(or (> 7 9) (< 1 9))` → `#t`
    - `(or (> 3 5) (= 1 9))` → `#f`
    - `(or (> 3 5) (+ 1 9))` → `Error`
- `not`: Logische Umkehrung
    - `(not (> 1 3))` → `#t`
    - `(not (> 3 1))` → `#f`
- `boolean=?`: Boolsche Gleichheit (2 Operanden)
    - `(boolean=? (> 3 1) (< 2 5))` → `#t` (beide `true`)
    - `(boolean=? (> 1 3) (< 5 2))` → `#t` (beide `false`)
    - `(boolean=? (> 3 1) (< 5 2))` → `#f` (`true` und `false`)

Mit dem Kurzschlussverfahren lassen sich Fehler vermeiden:

```
> (define x 0)
> (and (not (= x 0)) (< (/ 2 x) 1))
#f
```

Ist `x` gleich `0`, wird die Division nicht ausgewertet. Wird die Reihenfolge der `and`-Operanden umgedreht, gibt es einen Fehler:

```
> (define x 0)
```

```
> (and (< (/ 2 x) 1) (not (= x 0)))
; /: divison by zero
```

4.3.3 Prädikatfunktionen

Für Zahlen:

- `integer?`: ganze Zahl?
  - `(integer? (/ 4 2))` → `#t`
  - `(integer? (/ 4 3))` → `#f`
- `real?`: reelle Zahl?
  - `(real? (sqrt 2))` → `#t`
  - `(real? (sqrt -2))` → `#f`
- `rational?`: rationale Zahl?
  - `(rational? (/ 4 3))` → `#t`
  - `(rational? (sqrt -2))` → `#f`
- `complex?`: komplexe Zahl?
  - `(complex? (sqrt -2))` → `#t`
  - `(complex? (sqrt 2))` → `#f`
- `odd?`: ungerade?
  - `(odd? 3)` → `#t`
  - `(odd? 2)` → `#f`
- `even?`: gerade?
  - `(even? 4)` → `#t`
  - `(even? 3)` → `#f`
- `negative?`: negativ?
  - `(negative? -3)` → `#t`
  - `(negative? 3)` → `#f`
- `positive?`: positiv?
  - `(positive? 3)` → `#t`
  - `(positive? -3)` → `#f`
- `zero?`: null?
  - `(zero? 0)` → `#t`
  - `(zero? 0.0001)` → `#f`
- `exact?`: exakt?
  - `(exact? (sqrt 4))` → `#t`
  - `(exact? (sqrt 2))` → `#f`
- `inexact?`: nicht exakt?
  - `(inexact? (sqrt 2))` → `#t`
  - `(inexact? (sqrt 4))` → `#f`

Für Typen:

- `boolean?`
- `number?`

- `char?`
- `string?`
- `symbol?`
- `vector?`
- `prodedure?`
- `null?`
- `pair?`

Für Gleichheitsprüfung:

- `eq?`: prüft, ob es sich um die gleichen Objekte im Speicher handelt
- `eqv?`: wie `eq?`, prüft Zahlen auf Äquivalenz, nicht auf Identität
- `equal?`: wie `eqv?`, prüft zusammengesetzte Zypen auf Äquivalent, nicht auf Identität
- `char=?`: Gleichheit von Zeichen
- `string=?`: Gleichheit von Strings
- `null?`: Nullprüfung

Prädikate lassen sich mittels `define` definieren:

```
> (define (is-the-answer-to-everything? expr) (= expr 42))
> (is-the-answer-to-everything? 42)
#t
> (is-the-answer-to-everything? 35)
#f
```

4.3.4 Symbole

- für symbolische Namen (nicht Zeichenketten)
- Sequenz von Zeichen beginnend mit einem einfachen Anführungszeichen `'`
  - `foobar`, `'this-is-a-test`
- `symbol=?`-Prädikat: Gleichheit von Symbolen
  - `(symbol=? 'Foo 'Foo)` → `#t`
  - `(symbol=? 'Foo 'Bar)` → `#f`

4.3.5 Fallunterscheidung

Fallunterscheidung mit dem `cond`-Schlüsselwort:

```
(cond (<condition clause 1> <expression 1>)
      (<condition clause 2> <expression 2>)
      ...
      (else <expression n>))
```

- Auswertung von oben nach unten
- Mindestens ein Fall muss zu `true` auswerten

Beispiel (`grade.rkt`):

```
#lang racket

(define (grade score)
  (cond
    ((not (number? score)) "score is not a number")
    ((not (integer? score)) "score is not an integer")
    ((<= 90 score 100) "A")
    ((<= 75 score 89) "B")
    ((<= 65 score 74) "C")
    ((<= 55 score 64) "D")
    ((<= 45 score 54) "E")
    ((<= 30 score 44) "FX")
    ((<= 0 score 29) "F")
    (else "out of range [0,100]")
  )
)

(grade 101)
(grade 100)
(grade 50)
(grade 0)
(grade (/ 99 4))
```

Ausgabe:

```
$ racket -t grade.rkt
"out of range [0,100]"
"A"
"E"
"F"
"score is not an integer"
```

4.3.6 Selektion

Selektion mit dem `if`-Schlüsselwort:

```
(if (<test>) <then-expression> <else-expression>)
```

Beispiel (Absolutwert):

```
(define (absolute x) (if (< x 0) (- x) x))
```

Die Ausdrücke `<then-expression>` und `<else-expression>` werden nur bei Bedarf ausgewertetn (lazy evaluation).

4.4 Strukturen

Strukturen sind zusammengesetzte Datentypen aus Atomen und anderen Strukturen.

Definition:

```
(define-struct <typename>
    (<field 1> <field 2> ... <field n>))
```

Beispiel:

```
(define-struct address
    (firstname lastname street number zipcode city))
(define-struct point-2d
    (x y))
(define-struct point-3d
    (x y z))
```

`define-struct` erzeugt weitere Funktionen:

1. `make-<typename>`: Konstruktor
   - `(make-point-2d 3 5)` → `#<point-2d>`
2. `<typename>?`: Typen-Prädikat
   - `(point-2d? (make-point-2d 10 20))` → `#t`
3. `<typename>-<field>`: Selektor für jedes Feld
   - `(define q (make-point-3d 10 20 30)) (point-3d-z q)` → `30`

4.4.1 Anwendungsbeispiel: Punktabstand (Hypothenuse) berechnen

Die Struktur `point` definiert die zwei Felder `x` und `y`. Die Funktion `hypot` erwartet zwei `point`-Instanzen, von denen sie den Abstand berechnet:

```
#lang racket

(define-struct point (x y))

; hypot = sqrt((p.x - q.x)² + (p.y - q.y)²)
(define (hypot p q)
  (sqrt (+ (expt (- (point-x p) (point-x q)) 2)
           (expt (- (point-y p) (point-y q)) 2))))

(define a (make-point 5 7))
(define b (make-point 2 3))
(hypot a b)
; sqrt((5 - 2)² + (7 - 3)²) = sqrt(3² + 4²) = sqrt(9 + 16) = sqrt(25) = 5
```

Ein Punkt mit zwei Koordinaten steht auch mit der eingebauten Struktur `posn` (Position) zur Verfügung.

4.5 Listen

Rekursive Liste: Entweder leer, oder bestehend aus einem Element und einem Rest, der wiederum eine Liste ist. Listen können Elemente beliebiger (auch unterschiedlicher) Datentypen enthalten.

- Selektoren:
  - `first`/`car`: erstes Element
  - `rest`/`cdr`: Restliste
- Prädikate:
  - `list?`: Liste?
  - `cons?`: nicht leere Liste?
  - `empty?`: leere Liste?
- Funktionen:
  - `reverse`: Reihenfolge umkehren
  - `length`: Länge der Liste
  - `append`: zwei Listen aneinanderhängen

Beispiel:

```
> (define list0 empty)
> list0
'()
> (first list0)
'()
> (rest list0)
'()
> (list? list0)
#t
> (cons? list0)
#f
> (empty? list0)
#t

> (define list1 (cons 'a empty))
> list1
'(a)
> (first list1)
'a
> (rest list1)
'()
> (list? list1)
#t
> (cons? list1)
#t
```

```
> (empty? list1)
#f
> (empty? (rest list1))
#t

> (define list2 (cons 'a (cons 'b empty)))
> list2
'(a b)
> (first list2)
'a
> (rest list2)
'(b)
> (first (rest list2))
'b
> (car (cdr list2))
'b
> (cadr list2)
'b


> (reverse (list 'a 'b 'c 'd))
'(d c b a)
> (length (list 'a 'b 'c 'd))
4
> (append '('a 'b 'c) '('x 'y 'z))
'('a 'b 'c 'x 'y 'z)
```

```
list0: []
list1: [a|]->[]
list2: [a|]->[b|]->[]
```

`cons` vs. `list`:

```
> (cons "foo" (cons "bar" (cons "qux" empty)))
'("foo" "bar" "qux")
> (list "foo" "bar" "qux")
'("foo" "bar" "qux")
> '("foo" "bar" "qux")
'("foo" "bar" "qux")
```

Die `list`-Funktion wertet im Gegensatz zum Quote-Konstruktor `'` alle Argumente aus:

```
> '((+ 2 2))
'((+ 2 2))
> (list (+ 2 2))
'(4)
```

Strukturen in Listen:

```
; using a list like a struct
(define payroll
    (list
        (list "Hinz" 7500)
        (list "Kunz" 6900)))

; using a real struct
(define-struct wage (name salary))
(define payroll
    (list
        (make-wage "Hinz" 7500)
        (make-wage "Kunz" 6000)))
```

4.5.1 Rekursive Funktionen auf Listen

Summieren einer Liste:

```
(define (sum l)
  (cond
    ((empty? l) 0) ; Rekursionsbasis
    (else (+       ; Rekursionsanweisung
            (first l)
            (sum (rest l))))))

(sum '(1 2 3 4 5)) ; 15
```

Quadrieren aller Listenelemente:

```
(define (square-list l)
  (cond
    ((empty? l) empty)
    (else
      (cons (* (first l) (first l))
            (square-list (rest l))))))

(square-list '(1 2 3 4 5)) ; '(1 4 9 16 25)
```

Insertion Sort:

```
(define (insert v l)
  (cond
    ((empty? l) (list v))
    ((<= v (first l)) (cons v l))
    (else (cons (first l)
                (insert v (rest l))))))
```

```
(define (insertion-sort l)
  (cond
    ((empty? l) empty) ; Rekursionsbasis: leere Liste (per Definition sortiert)
    (else (insert (first l)
                  (insertion-sort (rest l))))))) ; Rekursionsanweisung

(insertion-sort (list 9 4 6 1)) ; '(1 4 6 9)
```

4.6 Rekursion

Strukturelle Rekursion:

- Zerlegen der Eingabedaten in ihre strukturellen Komponenten
  - Beispiel: Listen in Listenelemente zerlegen
- ein Aufruf pro Komponente
- Funktionsschema: erstes Element behandeln, erneuter Aufruf für den Rest

Beispiel (Maximalwert einer Liste finden):

```
(define (maximum l)
  (cond
    ((empty? (rest l)) (first l))
    (else (cond
            ((> (first l) (maximum (rest l))) (first l))
            (else (maximum (rest l)))))))

(maximum '(5 2 6 9 4)) ; 9
```

Problem: zwei Rekursionsaufrufe!

Alternative: Akkumulative Rekursion:

- Akkumulator zum Zwischenspeichern von Zwischenresultaten (Teillösungen)
- Endrekursiv (tail recursive): rekursiver Funktionsaufruf als letzter Berechnungsschritt (optimierbar)

```
(define (maximum l acc)
  (cond
    ((empty? l) acc)
    ((> (first l) acc) (maximum (rest l) (first l)))
    (else (maximum (rest l) acc))))

(maximum '(5 2 6 9 4 8 1 3 7) 0) ; 9
```

Beispiel (grösster gemeinsamer Teiler, naive Lösung):

```
(define (ggt a b)
  (ggt-internal a b (min a b)))
```

```
(define (ggt-internal a b divisor)
  (cond
    ((= divisor 1) 1)
    ((and (= (remainder a divisor) 0)
          (= (remainder b divisor) 0)) divisor)
    (else (ggt-internal a b (- divisor 1)))))

(ggt 318 54)
```

Problem: jeder Divisor von 1 bis `min(a,b)` wird durchprobiert (ineffizient).

ggT-Algorithmus von Euklid:

```
(define (ggt-euklid a b)
  (cond
    ((zero? b) a)
    (else (ggt-euklid b (remainder a b)))))

(ggt-euklid 318 54)
```

Generative Rekursion:

- generativer Schritt: Argumente werden bei jedem Aufruf neu berechnet (`remainder a b`)
- Ansatz: Teile und Herrsche (Divide & Conquer)
  - Problem in Teilprobleme aufteilen
  - kleine Probleme lösen (Trivialfall)
  - Teillösungen zu Gesamtlösung kombinieren

4.7 Funktionen höherer Ordnung

- Funktionen erster Ordnung: Führen Berechnungen (im weitesten Sinne) anhand von Daten aus
- Funktionen höherer Ordnung: Beschreiben allgemeine Berechnungen, die mittels Funktionsparameter spezifiziert werden

Beispiel (Grundrechenarten):

```
; spezifische Funktionen
(define (plus a b)
  (+ a b))
(define (minus a b)
  (- a b))
(plus 3 5) ; 8
(minus 8 3) ; 5
```

```
; allgemeine Funktion mit Operation als Parameter
(define (calc a b f)
  (f a b))
(calc 3 5 +) ; 8
(calc 8 3 -) ; 5
```

- `plus` und `minus` sind Funktionen *erster Ordnung*
- `calc` ist eine Funktion *höherer Ordnung*

In Scheme sind Funktionen Daten und *Werte erster Klasse* und können:

- als Parameter übergeben werden
- Rückgabewerte anderer Funktionen sein
- an Namen gebunden werden
- in Listen und Strukturen aufgenommen werden

Anwendungsbeispiele:

- `filter`: Filterprädikate auf Listen durchführen (Filterfunktion als Parameter)
- `map`: Mapping von Listenwerten (Anwenden einer Parameter-Funktion auf alle Listenelemente)
- `apply`: Reduktion von Listenwerten auf einen Wert (Listenelemente als Eingabeparameter für eine Parameter-Funktion)

Beispiel (`filter`, `map`, `apply`):

```
; filter
(define (positive? x)
  (> x 0))
(filter positive? '(-3 5 -2 1)) ; '(5 1)

; map
(define (square x)
  (* x x))
(map square '(1 2 3 4 5)) ; '(1 4 9 16 25)

; apply
(apply + '(1 2 3 4)) ; 10
```

Beispiel (`reduce`-Implementierung):

```
(define (reduce l f)
  (cond
    ((= 2 (length l)) (f (first l) (first (rest l))))
    (else (f (first l) (reduce (rest l) f)))))

(define (plus a b)
  (+ a b))
```

```
(reduce '(1 2 3 4) plus) ; 10

(define (bigger a b)
  (if (> a b)
      a
      b))

(reduce '(4 3 1 6 3) bigger) ; 6
```

4.8 Scoping

4.8.1 Lokaler Scope

Per `define` definierte Symbole (Variablen, Funktionen) sind global sichtbar. Mit `local` kann ein lokaler Scope definiert werden:

```
(local (<definition 1>
        <definition 2>
        ...
        <expression>))
```

Die Definitionsliste wird abgearbeitet, `<expression>` evaluiert und zurückgegeben. Der Ausdruck `<expression>` verwendet in der Regel die zuvor innerhalb von `local` definierten Symbole, welche von ausserhalb nicht zugreifbar sind. Beispiel:

```
(define (binom a b)
  (local
    (
     (define (square a) (* a a)) ; a², b²
     (define (double-product a b) (* 2 a b))) ; 2ab
    (+ (square a) (double-product a b) (square b)))) ; a²+2ab+b²

; (3+4)² = 3²+2*3*4+4² = 9+24+16 = 49
(binom 3 4)
```

Die Parameter `a` und `b` der Funktionen `square` und `double-product` haben nichts mit den Parametern `a` und `b` der Funktion `binom` zu tun (lexikalisches Scoping).

4.8.2 Lokale Variablen

Lokale Variablen können mithilfe einer Kombination von `local` und `define` erstellt werden:

```
(define (binom-local-vars a b)
  (local
    (
```

```
     (define x (* a a)) ; a²
     (define y (* 2 a b)) ; 2ab
     (define z (* b b))) ; b²
    (+ x y z)))

; (3+4)² = 3²+2*3*4+4² = 9+24+16 = 49
(binom-local-vars 3 4)
```

Die gleiche Definition kann auch kürzer mit `let` (nur für Variablen) ausgedrückt werden:

```
(let
 (
  (<var 1> <expr 1>)
  (<var 2> <expr 2>)
  ...
  (<var n> <expr n>)
  ))
```

Das Beispiel von vorher:

```
(define (binom-let a b)
  (let
    (
     (x (* a a))
     (y (* 2 a b))
     (z (* b b)))
    (+ x y z)))

; (3+4)² = 3²+2*3*4+4² = 9+24+16 = 49
(binom-let 3 4)
```

Bei `let` ist zu beachten, dass zunächst alle Ausdrücke (`expr 1` bis `expr n`) ausgewertet werden, bevor Zuweisungen an die Symbole (`var 1` bis `var n`) stattfinden:

```
(define (sevenfold x)
  (let
    (
     (double (* 2 x))
     (fourfold (* 2 double))) ; double: unbound identifier
    (+ fourfold double x))) ; 7x = 4x+2x+x

(sevenfold 2)
```

Die Form `let*` hingegen wertet die Definitionsliste von links nach rechts/oben nach unten aus (`expr 1` auswerten und an `var 1` zuweisen, dann `expr 2` auswerten und an `var 2` zuweisen usw.):

```
(define (sevenfold x)
```

```
  (let*
    (
     (double (* 2 x))
     (fourfold (* 2 double)))
    (+ fourfold double x))) ; 7x = 4x+2x+x

(sevenfold 2) ; 14
```

4.9 Anonyme Funktionen

Mit der Form `lambda` können anonyme Funktionen erstellt werden. Diese sind dann hilfreich, wenn eine Funktionsdefinition nur einmal in einem bestimmten Kontext benötigt wird, wie z.B. bei dieser `reduce`-Funktion, die eine Liste und einen Funktionsausdruck erwartet:

```
(define (reduce l f)
  (cond
    ((= 2 (length l))
     (f (first l) (first (rest l))))
    (else
      (f (first l)(reduce (rest l) f)))))

; named function
(define (plus a b) (+ a b))
(reduce '(1 2 3 4) plus)

; lambda expression
(reduce '(1 2 3 4) (lambda (a b) (+ a b)))
```

`lambda`-Ausdrücke können nicht rekursiv verwendet werden, da sie keinen Namen haben.

4.10 Seiteneffekte

Reine Funktionen haben keinen Status. Ihr Rückgabewert ist ausschliesslich von den Parametern und der Funktionsdefinition abhängig. Oft müssen in der Programmierung Zustände abgespeichert und aktualisiert werden. Dazu kann die `set!`-Form verwendet werden, welche den Wert einer Variablen überschreibt.

Beispiel (Counter mit Closure):

```
(define (counter current)
  (lambda ()
    (set! current (+ current 1)) ; override value
    current))

(define c (counter 0))
```

```
(c) ; 1
(c) ; 2
(c) ; 3
```

4.11 Sequenzen

Die `begin`-Form wertet eine Liste von Ausdrücken aus und gibt den Wert des letzten evaluierten Ausdrucks zurück. Damit lassen sich prozedurale Programmkonstrukte umsetzen:

```
(define (binom-seq a b)
  (begin
    (define result 0)
    (set! result (* a a)) ; a²
    (set! result (+ result (* 2 a b))) ; + 2ab
    (set! result (+ result (* b b))) ; + + b²
    result))

(binom-seq 3 4) ; 49
```

4.12 Praktische Tipps

Mit Racket eine Datei einlesen (`-t`) und deren Symbole interaktiv (`-i`) im REPL (read-eval-print-loop) verwenden (`interactive.rkt`):

```
#lang racket
(provide x) ; make x available in REPL!
(define x 3)
(define y 4)
```

Ausführung:

```
racket -it interactive.rkt
> x
3
> y
; y: undefined
```

5 Java 8

5.1 Interfaces

5.1.1 `default`-Methoden

Seit Java 8 können Interfaces auch Implementierungen vorgeben:

```java
interface Calculator {
    public default int calculate(int a, int b) {
        return a + b;
    }
}

// write custom implementation
Calculator multiplier = new Calculator() {
    public int calculate(int a, int b) {
        return a * b;
    }
};
System.out.println(multiplier.calculate(5, 4)); // 5 * 4 = 20

// use default implementation (since Java 8)
Calculator adder = new Calculator() {};
System.out.println(adder.calculate(5, 4)); // 5 + 4 = 9
```

Hintergrund: Wichtige Methoden für die Streams-API (`stream()`, `parallelStream()`) werden direkt auf dem Interface `Collection` angeboten, sodass keine bestehenden Interfaces in der Collections-API verändert werden mussten (Rückwärtskompatibilität: *defender methods*).

5.1.2 Mehrfachvererbung

Java unterstützt keine Mehrfachvererbung. Eine Java-Klasse kann aber mehrere Interfaces implementieren. Geben zwei implementierte Interfaces die gleiche `default`-Methode vor, muss die Methode mit dem zu verwendenden Interface qualifiziert werden:

```java
interface Adder {
    public default int calculate(int a, int b) {
        return a + b;
    }
}

interface Divider {
    public default int calculate(int x, int n) {
        return x / n;
    }
}

class AverageCalculator implements Adder, Divider {
    public int calculate(int a, int b) {
```

```
        return Divider.super.calculate(Adder.super.calculate(a, b), 2);
    }
}

System.out.println(new AverageCalculator().calculate(4, 6)); // 5
```

5.1.3 static-Methoden

Ab Java 8 können Interfaces auch statische Methoden definieren:

```
interface Calculator {
    public static int add(int a, int b) {
        return a + b;
    }
}

System.out.println(Calculator.add(4, 6)); // 10
```

5.2 Lambda-Ausdrücke

Mithilfe von Lambda-Expressions kann die Implementierung eines Interfaces vereinfacht werden:

```
// prior to Java 8: implement interface
List<Integer> numbers = Arrays.asList(3, 4, 1, 2);
Collections.sort(numbers, new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        return a - b;
    }
});
System.out.println(numbers); // [1, 2, 3, 4]


// since Java 8: lambda expression
List<Integer> moreNumbers = Arrays.asList(6, 8, 7, 5);
Collections.sort(moreNumbers, (a, b) -> a - b);
System.out.println(moreNumbers);
```

Die Datentypen des Rückgabewerts und der Argumente müssen nicht explizit angegeben werden, da sie vom Kontext abgeleitet werden können (*type inference*).

5.2.1 Funktionale Interfaces

Ein Lambda-Ausdruck implementiert implizit ein *functional interface*; ein Interface, das mit der Annotation `@FunctionalInterface` definiert ist. Ein funktionales Interface verfügt

über eine abstrakte Methode. Interface- und Methodenname sind dabei nur informativer Natur, da sie im Code nicht auftauchen. Das Package `java.util.function` gibt funktionale Interfaces vor. Diese sind von einem bestimmten Grundtyp:

- Consumer: keine Rückgabe
- Function: Rückgabe beliebigen Typs
  - Predicate: Rückgabe eines Wahrheitswerts (`boolean`)
- Operator: Rückgabe vom Argument-Typ
- Supplier: Rückgabe ohne Eingabeargument

Weiter gelten folgende Namenskonventionen:

- Präfix «Bi» und «Binary» (für Operator) bei zwei Argumenten:
  - `BiPredicate<T, U>`
  - `BinaryOperator<T>`
- Präfix «Unary» für Operator mit einem Argument:
  - `UnaryOperator<T>`
- Elementarer Datentyp als Prefix:
  - `DoublePredicate`
  - `IntegerConsumer`
- «To» für Funktionen mit unterschiedlichen Argument- und Rückgabetypen:
  - `IntToLongFunction`

5.2.2 Scoping

Lambda-Ausdrücke übernehmen den umschliessenden Scope, dürfen aber nur lesend auf Variablen in diesem zugreifen. Man sagt, die lokalen Variablen müssen *effectively final* sein:

```
int a = 1;
Arrays.asList(1, 2, 3).forEach(i -> {
    int b = a; // reading: allowed
    System.out.println(i + b);

    a = 2; // writing: compilation error!
    System.out.println(i + a);
});
```

5.2.3 Methoden-Referenzen

Ab Java 8 können Methoden mit dem `::`-Operator referenziert werden. Dadurch können bestehende Methoden wie Lambda-Ausdrücke verwendet werden. Es gibt vier Arten von Methoden-Referenzen:

1. statische Methode:
   - `System::currentTimeMillis`
2. Instanzmethode eines Objekts:

    - `System.out::println`
3. Instanzmethode einer Klasse:
    - `String::length`
4. Konstruktor:
    - `String::new`

Anwendungsbeispiel:

```
// print one element per line
Arrays.asList(1, 2, 3, 4).forEach(Sytem.out::println);
```

Methodenreferenzen können mit funktionalen Interfaces referenziert werden:

```
LongSupplier millis = System::currentTimeMillis;
Consumer<String> println = System.out::println;
ToIntFunction<String> length = String::length;
Supplier<String> newString = String::new;
```

5.3 Streams

Idee: Anwendung von Funktionen auf Datenstrukturen; implizite Iteration.

- `forEach`: Funktion für jedes Element aufrufen
- `filter`: Filterung der Liste anhand einer Prädikatfunktion zur Erstellung einer neuen Sequenz (Teilmenge)
- `map`: Funktion auf jedes Element einer Sequenz anwenden, um Abbildung der Sequenz zu erhalten
- `reduce`: Einzelnes Resultat aus allen Elementen produzieren

Umsetzung:

- für generische Streams:
    - `interface java.util.stream.Stream<T>`
- für primitive Datentypen:
    - `IntStream`
    - `LongStream`
    - `DoubleStream`

Stream erhalten:

- `Collection<T>.stream()` auf bestehender Collection
- `Arrays.stream(T[] array)`: für ein Array
- `Stream<T>.of(T... values)`: generische Factory-Methode
- `IntStream.range(int startInc, int endExc)`: Factory-Methode für Sequenzen

5.3.1 Intermediate/Terminal Operations

Es gibt zwei Arten von Stream-Operationen:

1. intermediate (lazy): erzeugt neuen Stream (`filter`, `map`)
2. terminal (eager): macht etwas anderes (`forEach`, `reduce`)

Intermediate-Operationen werden nicht direkt ausgewertet, sondern erst beim Aufruf der ersten Terminal-Operation. Beispiel:

```
Integer[] numbers = { 1, 2, 3, 4 };
int sumOfSquaredEvenNumbers = Arrays.stream(numbers)
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .reduce(0, (x, y) -> x + y);
```

(Da `reduce` immer mit zwei Werten arbeitet, benötigt es für den ersten Durchlauf einen Initialwert. Dieser sollte ein neutrales Element sein, z.B. `0` für eine Addition, `1` für eine Multiplikation und `""` für eine `String`-Konkatenierung.)

Auswertungsreihenfolge:

1. `filter(1)`
2. `filter(2)`
3. `map(2)=2²`
4. `reduce(0,2²)`
5. `filter(3)`
6. `filter(4)`
7. `map(4)=4²`
8. `reduce(2², 4²)`

Die Sequenz wird nicht mehrmals durchiteriert. Stattdessen führt der Aufruf einer terminal-Operation dazu, dass vorhergehende intermediate-Operationen evaluiert werden müssen. Dies führt zu einer besseren Performance (Optimierungen, Parallelisierung möglich) und zu einer einfacheren Implementierung (kein Buffern von Zwischenresultaten nötig).

Wird eine terminal-Operation auf einen Stream angewendet, gilt dieser anschliessend als konsumiert und darf nicht mehr verwendet werden.

5.3.2 Stateful/Stateless Intermediate Operations

- Bei stateless-Operationen hängt das Ergebnis nur vom jeweiligen Stream-Element ab.
  - `filter`
  - `map`
- Bei stateful-Operations wird ein zusätzlicher Zustand benötigt
  - `limit`
  - `distinct`
  - `substream`
  - `sorted`

Stateless-Operationen sind einfacher handhab- und parallelisierbar. Können `filter` und `map`

in beliebieger Reihenfolge ausgeführt werden, blockiert z.B. `sorted` den ganzen Stream, bis die Operation abgeschlossen ist.

5.3.3 Short-Circuiting Operations

Die Stream-Abarbeitung kann vor dem Erreichen des letzten Elements gestoppt werden:

- intermediate:
  - Begrenzung: `limit(long maxSize)`
  - Überspringen: `skip(long n)`
- terminal:
  - Prädikate: `anyMatch()/noneMatch()/allMatch()`
  - Selektoren: `findAny()/findFirst()`

5.3.4 Parallele Streams

Die Methode `Stream<T>.parallelStream()` gibt einen Stream zurück, dessen Abarbeitung nach Möglichkeit nebenläufig vonstatten geht. Parallele und sequenzielle Streams können ineinander umgewandelt werden, wozu das Interface `BaseStream` folgende Methoden definiert:

- `boolean isParallel()`
- `S parallel()`
- `S sequential()`

Bei parallelen Streams ist die Auswertungsreihenfolge nicht deterministisch!

5.3.5 Unendliche Streams

Da Streams *lazy evaluation* ermöglichen, sind Konstrukte wie unendlich lange Sequenzen möglich. Beispiel:

```
double sum = DoubleStream.generate(Math::random))
    .skip(10_000)
    .limit(5_000)
    .sum();
System.out.println("average: " + sum / 5_000);
```

Ein (theoretisch) unendlich langer Stream mit Zufallszahlen im Bereich `0..1` wird erstellt. Die ersten zehntausend Elemente werden übersprungen, dann wird der Stream auf eine Länge von fünftausend Elementen eingeschränkt, und die Elemente werden aufsummiert.

5.3.6 `Optional<T>`

Da Methode `Stream<T>.reduce()` auf einem leeren Stream kein Ergebnis erzeugen kann, wird nicht das eigentliche Resultat, sondern ein `Optional` zurückgeliefert. Wichtige Metho-

den von `java.util.Optional` sind:

- `boolean isPresent()`: gibt an, ob ein Wert vorhanden ist
- `T get()`: gibt den Wert zurück, falls er vorhanden ist
- `T orElse(T value)`: gibt den Wert (falls vorhanden) oder den Parameter (falls kein Wert vorhanden) zurück
- `T ifPresent(Consumer<T> c)`: führt den Consumer-Lambda-Ausdruck aus, falls ein Wert vorhanden ist

Beispiel:

```
List<Integer> list = Arrays.asList(1, 2, 3);
Optional<Integer> opt = list.stream().reduce((a, b) -> a + b);
if (opt.isPresent()) {
    System.out.println(opt.get());
}
System.out.println(opt.orElse(0));
opt.ifPresent(System.out::println);
```

5.4 Weitere Java-Neuerungen

5.4.1 Strings verketten

- `String.join(CharSequence delimiter, CharSequence... elements)`
  - Verkettet die Elemente zu einem durch `delimiter` getretten `String`
- `String.join(CharSequence delimiter, Iterable<? extends CharSequence> elements)`
  - Das gleiche mit einem Iterator
- `java.util.StringJoiner`: Strings per Builder-Pattern verketten
  - `StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`: Konstruktor
  - `add(CharSequence element)`: Elemente hinzufügen

Beispiel:

```
String a = String.join(" ", "Das", "ist", "ein", "Test");

List<String> l = Arrays.asList("Das", "ist", "ein", "Test");
String b = String.join(" ", l);

StringJoiner j = new StringJoiner(":", "(", ")");
j.add("Das");
j.add("ist");
j.add("ein");
j.add("Test");
```

5.4.2 Komparatoren

Neue Methoden auf `Comparator`:

- `static Comparator<T> naturalOrder()`: Erstellt einen `Comparator` zur natürlichen Sortierung
- `Comparator<T> reversed()`: Erstellt einen `Comparator` zur umgekehrt natürlichen Sortierung

Beispiel:

```
// natural order
Comparator<String> cmp = Comparator.naturalOrder();
cmp.compare("a", "b"); // -1
cmp.compare("b", "a"); // 1
cmp.compare("a", "a"); // 0

// reversed natural order
Comparator<String> rev = cmp.reversed();
rev.compare("a", "b"); // 1
rev.compare("b", "a"); // -1
rev.compare("a", "a"); // 0
```

- `Comparator<T> thenComparing(Comparator<? super T> other)`: Verkettet zwei `Comparators`
- `static Comparator<T> nullsLast()/nullsFirst()`: Definiert die Einordnung von `null`-Werten

Beispiel:

```
Comparator<String> fullCmp = Comparator
    .comparingInt(String::length)
    .thenComparing(String.CASE_INSENSITIVE_ORDER);
fullCmp.compare("foo", "goo"); // -1
fullCmp.compare("foobar", "foo"); // 1

Comparator<String> natural = Comparator.naturalOrder();

Comparator<String> nullsFirst = Comparator.nullsFirst(natural);
nullsFirst.compare(null, "b"); // -1
nullsFirst.compare("a", "b"); // -1
nullsFirst.compare("b", null); // 1

Comparator<String> nullsLast = Comparator.nullsLast(natural);
nullsLast.compare(null, "b"); // 1
nullsLast.compare("a", "b"); // -1
nullsLast.compare("b", null); // -1
```

5.4.3 `CompletableFuture`

Mithilfe von einem `CompletableFuture` lassen sich zeitaufwändige Aufgaben mithilfe von Lambda-Ausdrücken nebenläufig ausführen, ohne den Hauptthread zu blockieren. Durch die Kombination von `supplyAsync`, `thenApplyAsync` und `thenAcceptAsync` lassen sich Zwischenergebnisse einfach von einem Thread zum nächsten weitergeben.

1. `static CompletableFuture<T> supplyAsync(Supplier<T> supplier)`: Initialisierung, stellt «Problem» zur Verfügung (`Supplier`)
2. `CompletableFuture<T> thenApplyAsync(Function<? super T, ?extends U> fn)`: Berechnung, führt das Problem in eine Lösung über (`Function`)
3. `CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)`: Verwertung, verbraucht Lösung (`Consumer`)

Beispiel:

```java
import java.util.concurrent.CompletableFuture;

public class Main {

    // task1 -> "42" -> task2 -> "the answer is 42"
    public static void main(String[] args) {
        CompletableFuture<String> task1 = CompletableFuture.supplyAsync(() -> {
            sleep(1000);
            System.out.print("task1");
            return "42";
        });
        CompletableFuture<String> task2 = task1.thenApplyAsync((String s) -> {
            sleep(1000);
            System.out.print("task2");
            return "the answer is " + s;
        });
        task2.thenAcceptAsync((String s) -> {
            sleep(1000);
            System.out.print(s);
        });
        for (int i = 0; i < 10; i++) {
            System.out.print(".");
            sleep(500);
        }
        System.out.print("done");
    }

    private static void sleep(long millis) {
        try {
```

```
            Thread.sleep(millis);
        } catch (InterruptedException iEx) {
            iEx.printStackTrace();
        }
    }
}
```

Ausgabe:

```
..task1...task2..the answer is 42...done
```

5.4.4 Kollektoren

Die Methode `Stream.collect()` ist ein Spezialfall von `reduce()`. Sie reduziert die Elemente eines Streams mithilfe einer `Collector`-Implementierung. Die Klasse `java.util.stream.Collectors` bietet verschiedene Möglichkeiten, u.a.:

- `String Collectors.joining()`: Elemente als Strings konkatenieren
- `String Collectors.joining(CharSequence delimiter)`: Elemente mit einem Delimiter als Strings konkatenieren
- `List<T> Collectors.toList()`: Elemente in einer Liste sammeln
- `List<T> Collectors.toCollection(Supplier collectionFactory)`: Elemente in einer Collection sammeln (als `collectionFactory` dient z.B. `TreeSet::new`)
- `Map<K, V> Collectors.groupBy(Function<T, K> classifier)`: Elemente nach einem Kriterium gruppieren (als `classifier` dient eine Methode von `T` mit Rückgabetyp `K`)

Beispiel:

```
List<String> ugly = Arrays.asList("foo", "alice", "bar", "dilbert", "qux", "boss");
String nice = ugly.stream()
    .filter(s -> s.length() > 3)
    .map(s -> s.substring(0, 1).toUpperCase() + s.substring(1, s.length()))
    .collect(Collectors.joining(", "));
System.out.println(nice); // Alice, Dilbert, Boss
```

5.5 Funktionales Denken

Analogie: Sowie der Garbage Collector den Programmierer vom Memory Management befreit, befreit die funktionale Programmierung den Programmierer von Implementierungsdetails (implizite statt explizite Funktion, Zwischenspeichern von Ergebnissen in Variablen usw.). Der Programmierer arbeitet auf einer höheren Abstraktionsebene: Der Code spielt sich näher an der Problemebene als auf der Computer-Ebene (Variablen, Schleifen) ab. Viele Verarbeitungsprobleme lassen sich in drei Kategorien einteilen:

- filter: Einschränkung der Problemmenge
- transform/map: Überführung einer Eingabe in eine Ausgabe

- convert/reduce: Zusammenführen der Teillösungen zu einer Gesamtlösung

Wie man in der Physik zwischen potenzieller und kinetischer Energie unterscheidet, kann man in Java zwischen Streams und Collections unterscheiden: Ein Stream ist noch *keine* Sammlung von Elementen, kann aber schnell zu einer werden.

6 Compiler

Ein Compiler ist ein Übersetzungsprogramm, das:

- ein Programm in einer *Quellsprache* liest,
- es in ein äquivalentes Programm in einer *Zielsprache* übersetzt
- und dabei weder *Syntax* noch *Semantik* des Programms verfälscht.

Compiler erkennen Fehler und melden sie dem Benutzer:

- Syntaxfehler: `c = (a + b/2;` (fehlende schliessende Klammer)
- Semantikfehler: `x = y + z` (Variable y noch nicht definiert)

6.1 Kompilierungsphasen

Die Kompilierung läuft in zwei Phasen ab:

1. Analyse
    - lexikalische Analyse
        - Zerlegung des Quellcodes in seine Bestandteile
        - Erzeugung von Tokens daraus
    - syntaktische Analyse
        - Anwendung von Grammatikregeln auf Tokens (Beispiel: «sind `a` und `b` Ausdrücke, so ist auch `a*b` ein Ausdruck»)
        - Zwischendarstellung als Syntaxbaum (siehe Beispiel)
    - semantische Analyse
        - Namensprüfung (Namenskonventionen)
        - Typprüfung (Zuweisungen, Parameterübergaben, Rückgaben, Casting)
        - Identifikation von Operatoren und Operanden
        - Überprüfung des Kontrollflusses (Code erreichbar?)
        - Gültigkeitsbereiche (Eindeutigkeit, lexikalisches Scoping)
2. Synthese
    - Zwischencode-Erzeugung: Einfacher Zwischencode für virtuelle (konzeptionelle) Maschine
    - Code-Optimierung
        - Inlining von einfachen Funktionen
        - Eliminierung von Rekursionen
        - Plattformspezifische Optimierungen (je nach Befehlssatz)
    - Code-Erzeugung: Erzeugung des Maschinencodes

Die Analyse und Zwischencode-Erzeugung werden vom Compiler-Frontend übernommen. Das Compiler-Backend kümmert sich um die Optimierung und Erzeugung des finalen Codes.

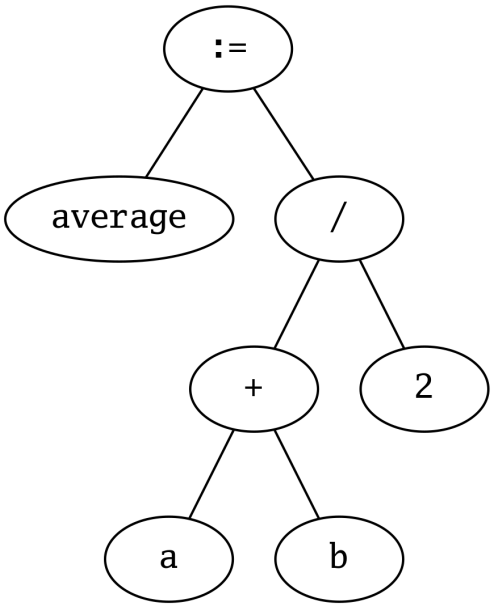


Abbildung 4: Syntaxbaum zum Ausdruck `average:=(a+b)/2`

6.2 Arten von Compilern

- Nativer Compiler: erzeugt Maschinencode für die Plattform, auf dem er selber läuft
- Cross-Compiler: erzeugt Maschinencode für eine andere Plattform
- Single-Pass-Compiler: erzeugt den Zielcode ohne Zwischenschritt
  - schnellerer Kompiliervorgang, dafür weniger optimierter Code
- Multi-Pass-Compiler: erzeugt den Zielcode in mehreren Schritten (siehe oben)
- Transcompiler: übersetzt Code von einer Hochsprache in eine andere
- Compiler-Compiler: Hilfsprogramm zur Generierung von Compiler-Teilen (z.B. `yacc`, ein Parser Generator)
- Just-in-time-Compiler (JIT-Compiler): übersetzt Code zur Laufzeit in Maschinencode, um die Ausführungsgeschwindigkeit gegenüber einem Interpreter zu steigern
- Compreter: Zwischencode-Erzeugung beim Kompiliervorgang, Interpretation zur Laufzeit
- Interpreter: Erzeugen keine Programme, sondern führen generierten Code direkt aus
  - Kombination mit JIT-Compilern möglich: `javac` erzeugt Bytecode, `java` optimiert zur Laufzeit

Vergleich	Compiler	Interpreter
Vorteil	einmal kompilieren, mehrmals ausführen; Laufzeitperformance	schnellerer Arbeitszyklus (try, fail, correct, retry)
Nachteil	mehrere Schritte bis zur Ausführung; aufwändige Fehlersuche	erneute Analyse bei jeder Ausführung; kein Executable

6.3 Bytecode und virtuelle Maschinen

- Bytecode ist vergleichbar mit dem Zwischencode eines traditionellen Compilers.
- Er wird beim Kompiliervorgang nicht weiter verarbeitet, sondern direkt auf einer virtuellen Maschine ausgeführt (oft mithilfe eines JIT-Compilers).
- Im Gegensatz zum traditionellen Zwischencode enthält Bytecode Typinformationen, Instruktionen zur Thread-Organisation und Metainformationen (Debug-Symbole); ist sehr kompakt und serialisierbar.
- Die virtuelle Maschine übernimmt Aufgaben wie Code-Optimierung (JIT-Compiler), Speicherorganisation (gemäss einem definierten Speichermodell), Thread- und IO-Organisation etc.

6.4 Formale Sprachen

- Sprache: Bildung von Sätzen aus einer definierten Menge von Zeichen unter Beachtung der Syntaxregeln.
- Alphabet (Σ): Menge von Zeichen, aus denen Wörter gebildet werden.
  - Terminales Alphabet (T): Menge von Zeichen, aus denen Sätze der Sprache gebildet werden (auch Atome, Morpheme, Token genannt).
    - `T={'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '-', '.'}`
  - Nichtterminales Alphabet (N): Menge von Metasymbolen zur Bildung syntaktischer Regeln.
    - `N={Zahl, GanzzahlTeil, BruchTeil}`
- Zeichenkette/Wort: Aneinanderreihung von Zeichen eines Alphabets.
  - Kann sowohl aus Elementen des terminalen wie auch des nichtterminalen Alphabets bestehen.

6.4.1 Grammatik nach Chomsky

Grammatik nach Chomsky: `G=(T,N,s,P)`

- `T`: terminales Alphabet
- `N`: nichtterminales Alphabet
- `s`: Startsymbol, Axiom
- `P`: Menge der Produktionen (Regeln)
  - Regeln: `[linke Seite] [Symbol] [rechte Seite]`
    - beide Seiten sind Zeichenketten des Alphabets

    * linke Seite: mindestens ein nichtterminales Symbol (wird durch die Anwendung der Regel ersetzt)
    * Symbole: `->`: wird abgeleitet nach, `::`, `::=`

Beispiel einer Grammatik:

```
T = { der, ein, Mann, Bier, trinkt }
N = { Satz, Subjekt, Prädikat, Objekt, Artikel, Verb, Substantiv }
S = Satz
P = {
    Satz       -> Subjekt Prädikat Objekt (1)
    Subjekt    -> Artikel Substantiv      (2)
    Objekt     -> Artikel Substantiv      (3)
    Prädikat   -> Verb                    (4)
    Substantiv -> Mann                    (5)
    Substantiv -> Bier                    (6)
    Verb       -> trinkt                  (7)
    Artikel    -> der                     (8)
    Artikel    -> ein                     (9)
}
```

- Ableitungen:
    - Linksableitung: das am weitesten links stehende nichtterminale Metasymbol wird untersucht
    - Rechtsableitung: das am weitesten rechts stehende nichtterminale Metasymbol wird untersucht
- Ableitbarkeit: Zeichenkette `t` ist aus Zeichenkette `s` ableitbar, wenn es eine Folge direkter Ableitung dieser Form gibt:
    - `s=>t`, falls `s->s1->s2->...->sn->t`
    - Beispiel: `Satz=>Der Mann trinkt das Bier`
    - direkte Ableitbarkeit: `t` durch Anwendung einer einzigen Regel aus `s` ableitbar
- Eine *Satzform* zu einer Grammatik G ist ein String, der aus einem Startsymbol (Axiom) ableitbar ist. Die Grammatik kann terminale und nichtterminale Symbole enthalten.
- Ein *Satz* ist eine Satzform, die nur terminal Symbole enthält.
- Eine *Sprache* ist die Menge der durch eine Grammatik G erzeugbaren Sätze.

Bildung eines Satzes:

Regel	Satzform
(1)	Satz
(2)	Subjekt Prädikat Objekt
(8)	der Substantiv Prädikat Objekt
(5)	der Mann Prädikat Objekt
(4)	der Mann Verb Objekt
(7)	der Mann trinkt Objekt

Regel	Satzform
(3)	der Mann trinkt Artikel Substantiv
(9)	der Mann trinkt ein Substantiv
(6)	der Mann trinkt ein Bier

Parsen eines Satzes:

Regel	Satzform
	der Mann trinkt ein Bier
(8)	Artikel Mann trinkt ein Bier
(5)	Artikel Substantiv trinkt ein Bier
(2)	Subjekt trinkt ein Bier
(7)	Subjekt Verb ein Bier
(4)	Subjekt Prädikat ein Bier
(9)	Subjekt Prädikat Artikel Bier
(6)	Subjekt Prädikat Artikel Substantiv
(2)	Subjekt Prädikat Objekt
(1)	Satz

6.4.2 Erweiterte Backus-Naur-Form (EBNF)

- BNF: Backus-Naur-Form (formale Syntax-Beschreibungssprache) von John Backus (Fortran) und Peter Naur (Algol)
- EBNF: Erweiterung von Niklaus Wirth (Pascal): wechselseitig rekursive Definition der syntaktischen Kategorien einer Programmiersprache
- Bausteine:
  - nichtterminale Symbole: syntaktische Kategorien
  - terminale Symbole: wörtliche Symbole der Sprache (in `' '`)
- BNF-Symbole:
  - `'='`: Definitionszeichen
  - `'|'`: Alternative
  - `''`: Kennzeichnung von Sprachsymbolen
  - `'ϕ'`: Leere Zeichenfolge
  - `'.'`: Ende einer Regel
- Zusätzliche EBNF-Symbole (Klammernpaare):
  - `()`: Gruppierung
  - `[]`: Option (`0..1`)
    - Alternative: `()?`
  - `{}`: optionale Wiederholung (`0..n`)
    - Alternative: `()*`
  - `<>`: beliebige Wiederholung (`1..n`)

        * Alternative: `()+`
- Aufbau der EBNF-Notation:
  1. Startregel
  2. sonstige Regeln
  3. terminale (Basiselemente) und nichtterminale (abgeleitete Konstrukte) Symbole
- EBNF-Regeln: `linke Seite = rechte Seite`
  - linke Seite: genau ein nichtterminales Symbol
  - rechte Seite:
    * terminale Symbole (in `' '`)
    * nichtterminale Symbole
    * leere Zeichenfolge `'ϕ'`
    * Alternativen (durch `'|'` gruppiert)
    * geklammerte Ausdrücke (nur EBNF)

Beispiel:

```
Ausdruck          = EinfacherAusdruck [ RelOp EinfacherAusdruck] .
EinfacherAusdruck = [ AddOp ] Term { AddOp Term } .
Term              = [ 'NOT' ] Faktor { MulOp Faktor } .
Variable          = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
AddOp             = '+' | '-' .
MulOp             = '*' | '/' .
RelOp             = '=' | '!=' | '<' | '<=' | '>' | '>=' .
```

Mögliche Ausdrücke (erste Regel als Startregel):

```
d
a + b
-(c/d)
(a*b) <> (c / f)
NOT (c/e)
```

6.5 Parsergeneratoren

Ein Parsergenerator erstellt aus einer Grammatik, die z.B. in EBNF vorliegt, ein Parser-Programm in einer bestimmten Sprache.

ANTLR steht für «ANother Tool for Language Recognition». Es bietet eine kompakte Sprache zur Formulierung von Sprachgrammatiken. Damit lassen sich lexikalische Scanner (Tokenizer) und Parser aus LL-Grammatiken erzeugen:

- LL: Abarbeitung von links nach rechts
- LL(k): Abarbeitung mittels Lookahead-Buffer von k Zeichen
- LL(*): Abarbeitung adaptiver Grammatiken

ANTLR bietet zwei Mechanismen zum Durchlaufen eines Syntaxbaums:

1. Listener Interface: reagiert auf Ereignisse, die vom Tree Walker ausgelöst werden (vgl. SAX).
    - `enter()`: Ereignis beim Eintritt in eine Regel
    - `exit()`: Ereignis beim Austritt aus einer Regel
2. Visitor Interface: bietet die Möglichkeit, den Syntaxbaum gemäss eigener Regeln abzuarbeiten.