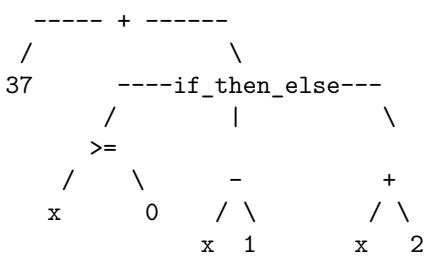
1 - Semantik von Ausdrücken

Termbaumdarstellung:



Präfixnotation: `+ 37 it_then_else >= x 0 - x 1 + x 2`

Postfixnotation: `37 x 0 >= x 1 - x 2 + if_then_else +`

Postfixnotation: `37 x 0 >= x 1 - x 2 + if_then_else +`

Belege hier x mit 37: `37 37 0 >= 37 1 - 37 2 + if_then_else +`

Stackmaschine:

```
   | 37 37 0 >= 37 1 - 37 2 + if_then_else +
=> 37 | 37 0 >= 37 1 - 37 2 + if_then_else +
=> 37 37 | 0 >= 37 1 - 37 2 + if_then_else +
=> 37 37 0 | >= 37 1 - 37 2 + if_then_else +
=> 37 true | 37 1 - 37 2 + if_then_else +
=> 37 true 37 | 1 - 37 2 + if_then_else +
=> 37 true 37 1 | - 37 2 + if_then_else +
=> 37 true 36 | 37 2 + if_then_else +
=> 37 true 36 37 | 2 + if_then_else +
=> 37 true 36 37 2 | + if_then_else +
=> 37 true 36 39 | if_then_else +
=> 37 36 | +
=> 73 |
```

Ruby wertet das `if_then_else` anders aus. In Ruby wird zunächst nur das erste Argument ausgewertet und dann in Abhängigkeit des Ergebnisses nur das zweite oder das dritte Argument ausgewertet. Es werden also nicht alle drei, sondern immer nur zwei Argumente ausgewertet.

Man sagt, `if_then_else` ist nicht strikt in seinem zweiten und dritten Argument. Dies ist insbesondere notwendig, um Rekusionen mittels `if_then_else` beenden zu können und Laufzeitfehler mittels `if_then_else` verhindern zu können.

2 - EBNF für Funktionsdefinitionen und Präzedenzen

```
Stm ::= Stm Stm
      | Var '=' Exp ';'
      | 'while' Exp 'do' Stm 'end' ';'
      | 'for' Var 'in' Exp '..' Exp 'do' Stm 'end' ';'
      | 'if' Exp 'then' Stm ['else' Stm] 'end' ';'
      | 'puts' '(' Exp ')' ';'
      | 'def' Var '(' [Var {',' Var}] ') Stm1 'end' ';'

Stm1 ::= Stm1 Stm1
```

```
      | Var '=' Exp ';'
      | 'while' Exp 'do' Stm1 'end' ';'
      | 'for' Var 'in' Exp '..' Exp 'do' Stm1 'end' ';'
      | 'if' Exp 'then' Stm1 ['else' Stm1] 'end' ';'
      | 'puts' '(' Exp ')' ';'
      | 'return' Exp ';'
```

Um das `return` nur innerhalb des Funktionsrumpfs zu erlauben, dupplizieren wir einfach die `Stm`-Regeln und fügen im Dupplikat (`Stm1`) eine Regel für `return` hinzu.

```
puts(-3**2)     # -9
puts((-3)**2)   # 9
puts(-(3**2))   # -9
```

Somit bindet `**` stärker als der unäre Operator `-`.

Der Operator `==` bindet schwächer als die Operatoren `+` und `*`, da ma in Ruby z.B. `3*4 == 10+2` ohne Klammern schreiben kann und der Ausdruck zu `true` auswertet. Überraschender Weise gibt es keine Links-/Rechts-Bindungspräzedenzen für `==`, was sich zeigt, wenn man den Ausdruck `true == false == false` auswertet. Man erhält einen Laufzeitfehler, während `(true == false) == false` und `true == (false == false)` beide zu `true` auswerten.

Um dies in der BNF abzubilden, müssen wir oberhalb von `+` einen weitere Ebene einziehen und beim Vorkommen von `==` in beiden Argumenten eine Ebene absteigen und so keine weiteren `==` ohne Klammern erlauben:

```
Exp  ::= Exp0 '==' Exp0 | Exp0
Exp0 ::= Exp0  '+' Exp1  | Exp1
Exp1 ::= Exp2 '*' Exp1 | Exp2
Exp2 ::= Num | '(' Exp ')'
```

3 - Tabellenartige Ausgabe zweidimensionaler Arrays

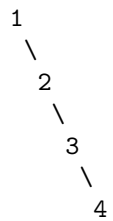
```
def longest_strings_in_columns(t)
  max_sizes = Array.new(t[0].size)
  for i in 0..t[0].size-1 do
    max_sizes[i] = 0
    for j in 0..t.size-1 do
      if t[j][i].length > max_sizes[i] then
        max_sizes[i] = t[j][i].length
      end
    end
  end
  return max_sizes
end


def pretty_print_table(t)
  sizes = longest_strings_in_columns(t)
  sum = 0
  for i in 0..sizes.size-1 do
    sum = sum + sizes[i]
  end
  sum = sum + sizes.size + 1
  puts("-" * sum)
  for i in 0 .. t.size-1 do
    for j in 0..t[0].size-1 do
      print("|" + t[i][j] + (" " * (sizes[j] - t[i][j].size)))
```

```
    end
    puts("|")
  end
  puts("-" * sum)
end
```

4 - Suchbaum

Die Funktion `insert!` steigt rekursiv im Suchbaum ab und fügt dann an der entsprechenden Position den Wert ein.

Im Worst-Case ist der Baum entartet und entspricht einer Liste, wie z.B. der folgende Suchbaum für die WErte 1 bis 4:



Fügt man nun einen größere Zahl ein, muss durch die gesamte Liste abgestiegen werden. Die Funktion verhält sich im Worst-Case also linear in der Anzahl der vorhandenen Elemente im Baum.

Dieser Fall ist auch der Best-Case. Wenn man in solch einen entarteten Baum einen kleineren Wert einfügt, ist dies direkt links unterhalb der Wurzel möglich und die Best-Case Laufzeit ist konstant.

Im Average-Case ist der Baum (halbwegs) ausgeglichen, weshalb ein Absteigen nur über logarithmisch viele Schritte notwendig ist. Die Average-Case Laufzeit ist also logarithmisch in der Zahl der Elemente im Baum.

```
def lookup(t,n)
  while t!=[] && t[1]!=n do
    if t[1] > n then
      t = t[0]
    else
      t = t[2]
    end
  end
  return t!=[]
end
```

```
def lookup_rek(t,n)
  if t == [] then
    return false
  elsif t[1]==n then
    return true
  elsif t[1] > n then
     lookup_rek(t[0], n)
  else
     lookup_rek(t[2], n)
  end
end
```

5 - Physikalische Größen

```
class Measurement

  def initialize(wert, zaehler, nenner)
    @wert    = wert.to_f()
    @zaehler = zaehler
    @nenner  = nenner
  end


  ##### GETTERS

  def get_wert()
    return @wert
  end

  def get_zaehler()
    return @zaehler
  end

  def get_nenner()
    return @nenner
  end


  ##### ARITHMETICS

  def add(m)
    # Schauen, ob Masseinheiten uebereinstimmen
    compatible_units = compatible_units(@zaehler, m.get_zaehler) &&
                       compatible_units(@nenner, m.get_nenner);

    if compatible_units then
      return Measurement.new(@wert + m.get_wert(), @zaehler, @nenner)
    else
      return nil
    end
  end

  def multiply(m)
    new_wert = @wert * m.get_wert()
    new_zaehler = @zaehler + m.get_zaehler()
    new_nenner = @nenner + m.get_nenner()

    return Measurement.new(new_wert, new_zaehler, new_nenner)
  end

  def divide(m)
    new_wert = @wert / m.get_wert()
    new_zaehler = @zaehler + m.get_nenner()
    new_nenner = @nenner + m.get_zaehler()

    return Measurement.new(new_wert, new_zaehler, new_nenner)
  end
```

```
  ##### HELPERS

  def show_measurement()
    if @nenner == "" then
      puts(@wert.to_s + " " + @zaehler)
    else
      puts(@wert.to_s + " (" + @zaehler + ")/(" + @nenner + ")")
    end
  end

  def compatible_units(s1, s2)
    if s1.length() == s2.length()
      s2 = s2.clone()

      for s1i in 0..s1.length() do
        s2i = 0;

        while s2i < s2.length() && s2[s2i] != s1[s1i] do
          s2i = s2i + 1;
        end

        if s2i < s2.length() then
          s2[s2i, 1] = ""
        end
      end

      return s2 == ""
    else
      return false;
    end
  end

end


##### TESTPROGRAMM

s = Measurement.new(50, "m", "")
s.show_measurement()
t = Measurement.new(2, "s", "")
t.show_measurement()

v = s.divide(t)
v.show_measurement()

v2 = Measurement.new(5, "m", "s")
v3 = v2.add(v)
v3.show_measurement()
```

6 - Mergesort

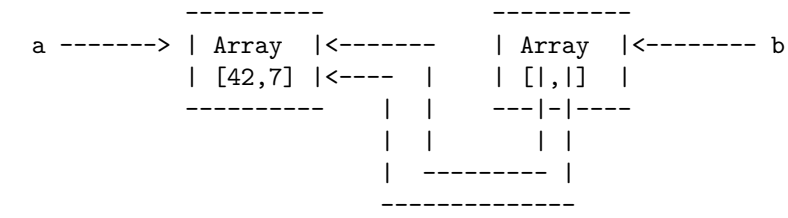
```
def merge(a1,a2)
  b = Array.new(a1.size+a2.size)
  i1 = 0
```

```
  i2 = 0
  while i1 < a1.size && i2 < a2.size do
    if a1[i1] < a2[i2] then
      b[i1+i2] = a1[i1]
      i1 = i1 + 1
    else
      b[i1+i2] = a2[i2]
      i2 = i2 + 1
    end
  end
  while i1 < a1.size do
    b[i1+i2] = a1[i1]
    i1 = i1 + 1
  end
  while i2 < a2.size do
    b[i1+i2] = a2[i2]
    i2 = i2 + 1
  end
  return b
end

def merge_sort(a)
  b = Array.new(a.size)
  for i in 0..a.size-1 do
    b[i] = [a[i]]
  end

  while b.size > 1 do
    i = 0
    while i < b.size-1 do
      b[i,2] = [merge(b[i],b[i+1])]
      i = i + 1
    end
  end
  return b[0]
end
```

7 - Mutierende und nicht mutierende Funktion

Die Zuweisung `a = [42,7]` legt eine neues Array-Objekt `[42,7]` im Speicher an. Die Variable `a` verweist auf diese Objekt.

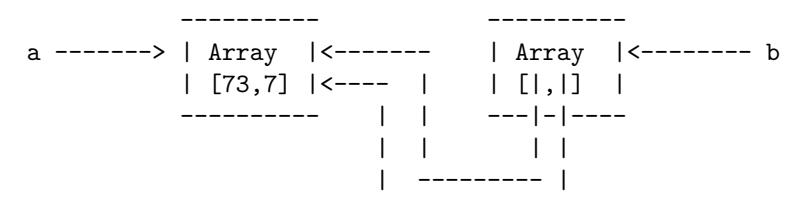
Die Zuweisung `b = [a,a]` legt ebenfalls ein Array-Objekt im Speicher an. Die Einträge in diesem Objekt sind jeweils das Onjekt, auf welches `a` verweist, also zwei mal die zuvor angelegten Objekte. Der Speicher sieht zu diesem Zeitpunkt also wie folgt aus:



Wenn man es ganz genau nimmt, sind auch 42 und 7 jeweils eigene Objekte, auf welche das erste Array-Objekt verweisen. Dies spielt für das Programm verhalten aber keine Rolle, weshalb wir dies vernachlässigen können.

In der nächsten Anweisung wird zunächst mittels `b[0]` das erste Array-Objekt (auf welches auch `a` verweist) selektiert, und dann mittels des mutierenden Methodenaufrufs (`[0]=73`) dieses Array so mutiert, dass der Eintrag an der Position `0` nun auf ein neues Objekt `73` verweist. Die genauen Methodenaufrufe dieser Anweisung sehen wie folgt aus: `b.[](0).[]=(0,73)`.

Der Speicher sieht dann wie folgt aus:



Das Array-Objekt, auf welches `b` verweist wurde gar nicht verändert. Da aber das ersten Array-Objekt verändert wurde und dieses in beiden Einträgen im zweiten Array-Objekt referenziert wird, sind die Änderungen, in beiden Array-Einträgen sichtbar. Außerdem natürlich auch bei der Ausgabe von `a`.

Die Funktion `improve` soll alle vorkommenden Werte `42` in einem Array durch den Wert `73` ersetzen. Hierbi soll das Array nicht mutiert werden, sondern ein neues Array zurück gegeben werden. Dies wird dadurch erreicht, dass anstelle der mutierenden Methode []= eine Zuweisung verwendet wird und mittels der nicht mutierenden Methode `+` ein neues Array konstruiert wird.

Allerdings wird in dem Fall, dass keine `42` im Array vorkommt keine Kopie des Arrays angelegt, sondern das Array selber zurück gegeben. Für Arrays gibt es auch mutierende Methoden. Wird später eine solche Methode aufgerufen, kann es zu einem merkwürdigem Programmverhalten kommen, weshalb wir in der Vorlesung die Konvention vorgegeben haben, dass nicht-mutierende Methoden (und natürlich auch Funktionen) ihr Argument in diesem Fall auf jeden fall kopieren müssen.

Das folgende Code-Fragment veranschaulicht die Problematik:

```
a = [1,2,3]  # oder
#a = [42,4]

b = improve(a)

a[0] = 0

p(b)
```

Mit obigem Code würde dieses Programm für den Fall `a = [1,2,3]` die Ausgabe `[0,2,3]` erzeugen und für den Fall `a=[42,4]` die Ausgabe `[73,4]`. Die Mutation des Arrays in `a` mittels `a[0] = 0` verändert also einmal `b` mit und einmal nicht.

Korrigieren läßt sich die nicht mutierend Funktion `improve` z.B. wie folgt:

```
def improve(a)
  for i in 0..a.size-1 do
    if a[i] == 42 then
      a = a[0,i] + [73] + a[i+1,a.size]
    end
  end
  return a.clone
end
```