Informatik I

8. Programmentwicklung und Einsatz rekursiver Datenstrukturen

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

12. November 2013

Programmentwicklung

Programmentwicklung: Das Problem verstehen

Wie entstehen Programme?

1. Identifikation eines Problems (im Informatiksinne):
   - Was soll durch den Computer gelöst werden?
2. Formale Spezifikation des Problems:
   - Was ist Eingabe? Was ist die Ausgabe?
   - Welches Verhalten soll erzeugt werden?
   - Erstellung eines Pflichtenhefts

Programmentwicklung: Umsetzung

3. Programmentwurf:
   - In welche (möglichst unabhängige) Einzelaufgaben lässt sich die Aufgabe zerlegen?
   - Welche Datenstrukturen sind geeignet, die Problemstellung zu repräsentieren?
   - Wie können wir das Programm schrittweise erstellen (und testen?)
4. Implementation:
   - Erstelle Programmteile für Einzelaufgaben
   - Dokumentiere die Schnittstellen!
   - Teste die Einzelteile auf ihre Funktionalität
   - Integriere alle Einzelteile und teste auf Gesamtfunktionalität
   - Dokumentiere die Schnittstellen nach außen (Fortschreibung Pflichtenheft)
5. Oft ist es notwendig, zu einem früheren Schritt zurück zu gehen

Programmentwicklung: Prototyping

- Statt alles vom Anfang bis zum Ende zu durchlaufen, erstelle möglichst *schnell* einen Prototyp – mit eingeschränkter Funktionalität
- Gut um ...
  - um mit dem „Kunden“ über die Anforderungen und die Funktionalität zu diskutieren,
  - um eine Idee vom Problem und wie man es lösen könnte zu bekommen.
- Schlecht, da
  - unter Umständen eine tief gehende Analyse nicht erfolgt,
  - der Kunde u.U. einen falschen Eindruck vom Projektstand bekommt.
- Python eignet sich auf Grund seiner Struktur sehr gut zum *Prototyping*

Fallstudie: Taschenrechner

Fallstudie: Taschenrechner

1. Man weiß mittlerweile, dass Sie Informatik studieren und Programmieren lernen.

Fallstudie: Taschenrechner

1. Man weiß mittlerweile, dass Sie Informatik studieren und Programmieren lernen.
2. Irgendwann am sonntäglichen Frühstückstisch: „Könntest Du mir nicht *mal eben* ein kleines Taschenrechnerprogramm schreiben?“

Fallstudie: Taschenrechner

1. Man weiß mittlerweile, dass Sie Informatik studieren und Programmieren lernen.
2. Irgendwann am sonntäglichen Frühstückstisch: „Könntest Du mir nicht *mal eben* ein kleines Taschenrechnerprogramm schreiben?“
3. Auch das Vorschieben des Abwasches und der noch nicht gelösten Übungsaufgaben bringt nichts.

Fallstudie: Taschenrechner

1. Man weiß mittlerweile, dass Sie Informatik studieren und Programmieren lernen.
2. Irgendwann am sonntäglichen Frühstückstisch: „Könntest Du mir nicht *mal eben* ein kleines Taschenrechnerprogramm schreiben?“
3. Auch das Vorschieben des Abwasches und der noch nicht gelösten Übungsaufgaben bringt nichts.
4. „Was möchtest du denn haben?“

Fallstudie: Taschenrechner

1. Man weiß mittlerweile, dass Sie Informatik studieren und Programmieren lernen.
2. Irgendwann am sonntäglichen Frühstückstisch: „Könntest Du mir nicht *mal eben* ein kleines Taschenrechnerprogramm schreiben?“
3. Auch das Vorschieben des Abwasches und der noch nicht gelösten Übungsaufgaben bringt nichts.
4. „Was möchtest du denn haben?“
5. „Ich würde gerne Zahlen addieren, multiplizieren und so weiter. Und bitte keine grafische Oberfläche!“

Die input-Funktion

- Wir schreiben schnell einen Prototyp und schauen dann ...
- Wie bekommt man Daten in den Python-Interpreter?
- Die input-Funktion erwartet als Argument einen String, der im Shell-Fenster ausgegeben wird und liefert als Ergebnis den vom Benutzer eingegeben String:

Python-Interpreter

```
>>> input('Dein Eingabe:')
Deine Eingabe:
```

Die input-Funktion

- Wir schreiben schnell einen Prototyp und schauen dann ...
- Wie bekommt man Daten in den Python-Interpreter?
- Die input-Funktion erwartet als Argument einen String, der im Shell-Fenster ausgegeben wird und liefert als Ergebnis den vom Benutzer eingegeben String:

Python-Interpreter

```
>>> input('Dein Eingabe:')
Deine Eingabe:blau
blau
```

Multi-Statement-Zeilen

- Es ist möglich, mehrere Anweisungen in einer Zeile getrennt durch Semikolon zu schreiben oder direkt nach dem Doppelpunkt ...
- ... sollte dies aber nur *sehr sparsam* verwenden
- Höchstens bei Einzeilern mit Körpern, die aus einer Anweisung bestehen:

Multi-Statement-Zeilen

```
#### Lieber nicht:
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
#### Geht gar nicht:
if foo == 'blah': do_one_thing(); do_two()
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()
```

Ein erster Versuch: Das Programm

Taschenrechner, die Erste

```
while True:
    op1 = input("Operand:  ")
    if (not op1): break
    op1 = int(op1)
    opa = input("Operator: ")
    op2 = int(input("Operand:  "))
    if opa == "+":
        print(op1+op2)
    elif opa == "-":
        print(op1-op2)
    elif opa == "*":
        print(op1*op2)
    else:
        print(op1//op2)
```

Vor- und Nachteile

- Vorteile:

Vor- und Nachteile

- Vorteile:
  1. Schnell eingetippt
  2. Einfach zu verstehen
  3. Einfach zu bedienen
- Nachteile:

Vor- und Nachteile

- Vorteile:
  1. Schnell eingetippt
  2. Einfach zu verstehen
  3. Einfach zu bedienen
- Nachteile:
  1. Unkommentiert
  2. Keine Fehlerbehandlung
     - Kryptische Python-Fehlermeldungen
     - Merkwürdige Ergebnisse bei Angabe eines „falschen“ Operators
- Rückmeldung des Kunden:

Vor- und Nachteile

- Vorteile:
  1. Schnell eingetippt
  2. Einfach zu verstehen
  3. Einfach zu bedienen
- Nachteile:
  1. Unkommentiert
  2. Keine Fehlerbehandlung
     - Kryptische Python-Fehlermeldungen
     - Merkwürdige Ergebnisse bei Angabe eines „falschen“ Operators
- Rückmeldung des Kunden:
  1. „Zu umständlich“
  2. „Ich würde gerne alles in eine Zeile schreiben!“

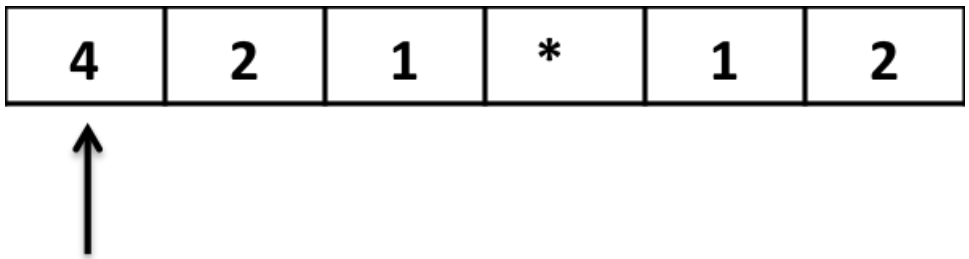
Ein zeilenbasierter Taschenrechner

- Eine gesamte Zeile einlesen.
- Falls Leerzeile, dann beenden
- Werte die Zeile aus und drucke das Ergebnis

Hauptprogramm

```
while True:
    line = input("Ausdruck: ")
    if not line:
        break
    print("Resultat:", eval_string(line))
print("Tschuess")
```

Evaluierung der Zeile



- Lese Zeichen für Zeichen und baue die erste Zahl auf (`ops[0]`).
- Wenn ein Zeichen kommt, das wie ein Operator aussieht, merke es dir (`op`).
- Danach lese die weiteren Ziffern (`ops[1]`).
- Wende den Operator an.

Die Auswertefunktion: Teil I

eval_line Anfang

```
def eval_string(expr):
    # Lese die Zeile und isoliere Operanden und Operatoren
    ops = [0, 0]
    i = 0
    for ch in expr:
        if ch >= '0' and ch <= 9:
            ops[i] = 10*ops[i] + ord(ch) - ord('0')
        elif ch in ['+', '-', '*', '/' ]:
            op = ch
            i = 1
    # Werte jetzt den Ausdruck aus
    ...
```

Die Auswertefunktion: Teil II

eval_line Schluss

```
    ...
    if ch == '+':
        return ops[0] + ops[1]
    elif ch == '-':
        return ops[0] - ops[1]
    elif ch == '*':
        return ops[0] * ops[1]
    elif ch == '/':
        return ops[0] // ops[1]
    else:
        return None
```

Was könnte besser gemacht werden?

1. Es stecken noch 2 Fehler im Code
2. Kommentare!
3. Leerzeichen in Operanden bemerken
4. Illegale Zeichen als Fehler markieren
5. Mehrere Operatoren als Fehler markieren

Die Kundenwünsche . . .

1. Der Kunde ist mit der neuen Version (nachdem sie korrigiert wurde) *viel* zufriedener.
2. Das Problem mit den Leerzeichen und den illegalen Zeichen merkt er gar nicht.
3. Eigentlich wäre es schöner, wenn man mehr als einen Operator eingeben könnte.

Kettenausdrücke

Wie kann man `eval_string` so *erweitern*, dass die Funktion auch mit mehreren Operatoren umgehen kann?

- Immer wenn ein *neuer* Operator gelesen wird, die vorhandenen beiden Werte verknüpfen und wieder in `ops[0]` speichern.
- Am Ende dann die beiden vorhandenen Werte verknüpfen.

Kettenausdrücke auswerten, Teil I

eval_string 1. Teil

```
def eval_string(expr):
    ops = [0, 0]
    i = 0
    op = ''
    for ch in expr:
        if ch >= '0' and ch <= '9':
            ops[i] = 10*ops[i] + ord(ch) - ord('0')
        elif ch in ['+', '-', '*', '/' ]:
            if op != '':
                if op == '+':
                    ops[0] = ops[0] + ops[1]
                elif op == '-':
                    ops[0] = ops[0] - ops[1]
                ...
```

Kettenausdrücke auswerten, Teil II

eval_string 2.Teil

```
                ...
                elif op == '*':
                    ops[0] = ops[0] * ops[1]
                elif op == '/':
                    ops[0] = ops[0] // ops[1]
                else:
                    return None
                ops[1] = 0
            op = ch
            i = 1
    ...
```

Kettenausdrücke auswerten, Teil III

eval_string 3.Teil

```
    ...
    if op == '':
        return ops[0]
    elif op == '+':
        return ops[0] + ops[1]
    elif op == '-':
        return ops[0] - ops[1]
    elif op == '*':
        return ops[0] * ops[1]
    elif op == '/':
        return ops[0] // ops[1]
    else:
        return None
```

Spaghetti-Code

- Von Spaghetti-Code spricht man, wenn in einem Programmabschnitt viele und unübersichtliche Kontrollstrukturen dazu führen, dass ein Leser nicht den Kontrollfluß nachverfolgen kann
- Entsteht speziell
  - beim *Losprogrammieren* von Anfängern,
  - und bei nachträglichen Änderungen.
- Wichtig: Isolierbare Teile möglichst auch isolieren!
- Doppelungen von Code vermeiden! (Cut & Paste-Programmierung)

→ *Refactoring*

Die Operator-Auswertungsfunktion

eval_bin_expr

```
def eval_bin_expr(op,op1,op2):
    if op1 == None or op2 == None:
        return None
    elif op == '':
        return op1
    elif op == '+':
        return op1 + op2
    elif op == '-':
        return op1 - op2
    elif op == '*':
        return op1 * op2
    elif op == '/':
        return op1 // op2
    else:
        return None
```

Die String-Auswertungsfunktion

eval_string

```
def eval_string(expr):
    ops = [0, 0]
    i = 0
    op = ''
    for ch in expr:
        if ch >= '0' and ch <= '9':
            ops[i] = 10*ops[i] + ord(ch) - ord('0')
        elif ch in ['+', '-', '*', '/' ]:
            if op != '':
                ops[0] = eval_bin_expr(op,ops[0],ops[1])
                ops[1] = 0
            op = ch
            i = 1
    return eval_bin_expr(op,ops[0],ops[1])
```

Ein Test

Python-Interpreter

```
>>>
Ausdruck:  6*7+3
Resultat:  45
Ausdruck:  1+a-9
Resultat:  -8
Ausdruck:  (3+4)*6
Resultat:  42
Ausdruck:  6*(3+4)
Resultat:  22
Ausdruck:
>>>
```

Das Problem mit dem kleinen Finger

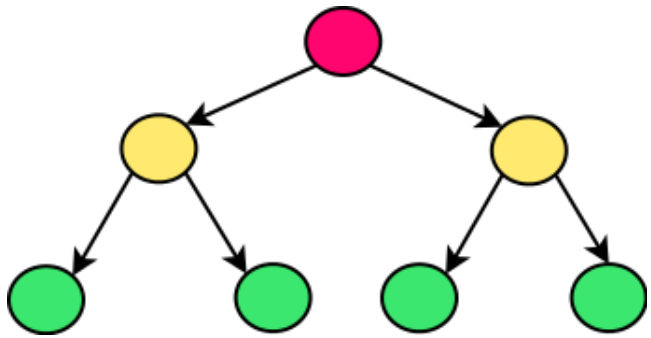
- Eigentlich wäre es doch ganz schön, wenn man ganz normale airthmetische Ausdrücke eingeben kann, die normal ausgewertet werden, d.h. als Eingaben sind zugelassen:
  1. natürliche Zahlen
  2. die normalen binären Operatoren `*`, `/`, `+`, `-`, wobei Punkt vor Strichrechnung geht
  3. Klammern, um die Operatorpräzedenz zu überschreiben
  4. Nicht wohl-geformte Ausdrücke sollten `None` als Ergebnis geben.
- Eigentlich bewegen wir uns hier in die Richtung Compilerbau, Parsing, usw., aber wir wollen die Herausforerung aufgreifen.

Entwurfsidee

- Wir transformieren die Eingabe in eine Tokenliste, die nur aus bedeutungstragenden lexikalischen Einheiten besteht.
- Dann erzeugen wir rekursiv schrittweise eine Baumstruktur, die die arithmetischen Teilausdrücke enthält.
- Zum Schluss lassen wir einen Auswerter rekursiv über diesen Baum laufen, um den Wert des Ausdrucks zu berechnen.

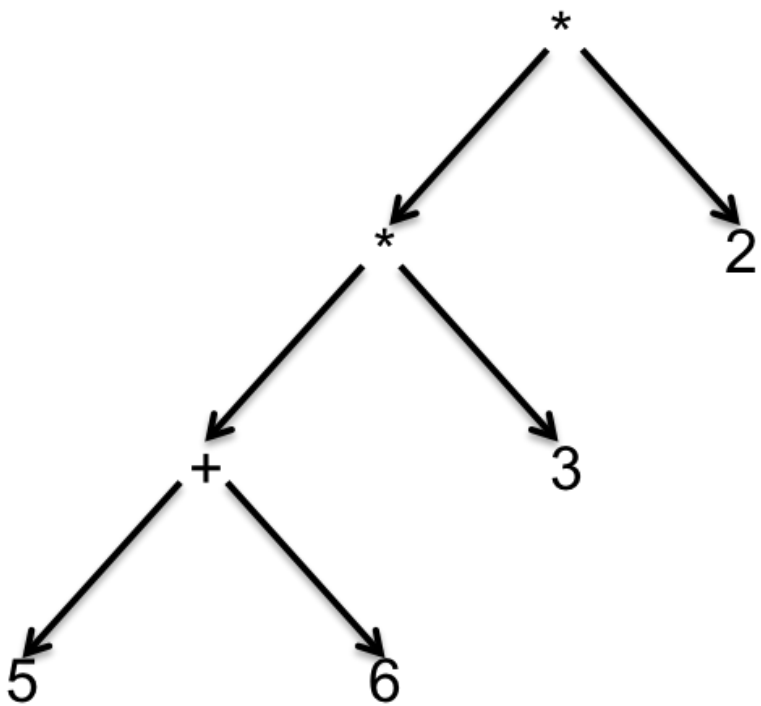
Exkurs: Bäume in der Informatik - Definition

- (Gewurzelte) Bäume sind in der Informatik allgegenwärtig
- Rekursive Definition: Ein Baum ist entweder leer oder besteht aus einer Wurzel mit $0 \ldots n$ Teilbäumen $t_1, \ldots, t_n$
- Gezeichnet werden sie meistens mit der Wurzel nach oben.
- Jede Wurzel eines Teilbaums heißt Knoten; Knoten ohne Teilbäume heißen Blätter; Knoten, die keine Blätter und keine Wurzel des Gesamtbaums sind, heißen innere Knoten.
- Knoten können Markierungen besitzen.



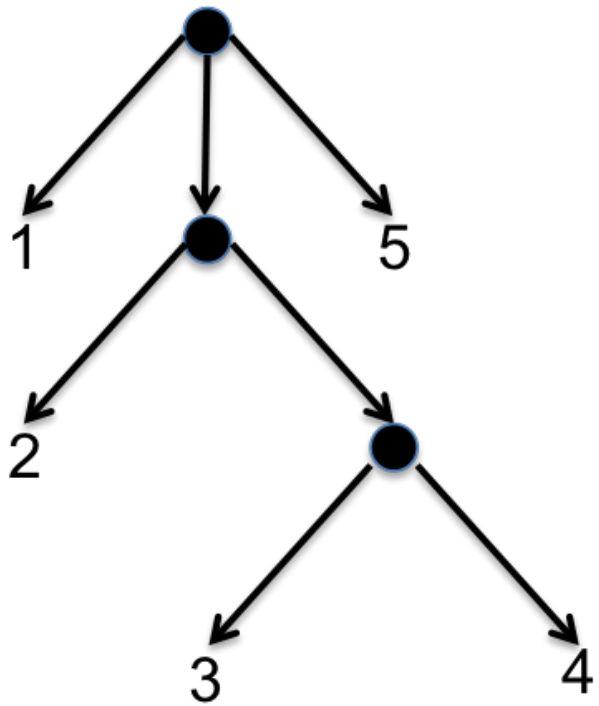
Exkurs: Bäume in der Informatik - Darstellung airthmetischer Ausdrücke

- Bäume können arithmetische (und andere) Ausdrücke so darstellen, dass ihre Auswertung eindeutig (und einfach durchführbar) ist
- Beispiel: $(5 + 6) * 3 * 2$



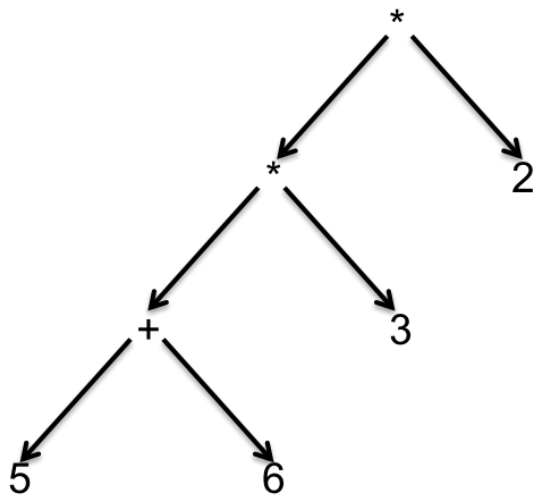
Listen und Tupel als Bäume

- Man kann jede Liste und jedes Tupel als Baum verstehen, bei dem die Liste/das Tupel selbst der Knoten ist und die Elemente die Teilbäume sind.
- Beispiel: `[1, [2, [3, 4]], 5]`



Markierte Bäume als Listen oder Tupel darstellen

- Bäume wie unser arithmetischer Baum haben Knotenmarkierungen
- Dies wird mit Hilfe einer Liste oft dadurch repräsentiert, dass die Markierung erstes Element der Liste wird (Tupel entsprechend):
  [<Markierung>,<linker Teilbaum>,<rechter Teilbaum>]



- Unser arithmetischer Baum könnte so dargestellt werden:
  `[ '*', [ '*', ['+', 5, 6], 3], 2]`

Unser Plan

- In der Tokenliste werden zuerst die geklammerten Ausdrücke zu Unterlisten zusammen gefasst (rekursiv!).
- Dann fassen wir von links nach rechts jeweils einen Operanden, einen Multiplikationsoperator, und einen weiteren Operanden zu einem Teilbaum zusammen. Dies wird wiederholt, solange es möglich ist (auch rekursiv).
- Schließlich wird das auch für die additiven Operatoren gemacht.

→ Wir erhalten eine Liste, die folgende Struktur (rekursiv) hat, falls der String ein wohlgeformter Ausdruck war:

  - Entweder sie besteht aus drei Elementen, woebi das erste ein Operator und die beiden anderen Operanden sind, die die gleiche Struktur haben, oder
  - sie besteht aus einem Element, das eine Liste mit der gleichen Struktur ist, oder
  - es handelt sich um eine Zahl

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$
2. Tokenliste: ['(', '5', '+', '6', ')', '*', '3', '*', '2']

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$
2. Tokenliste: ['(', '5', '+', '6', ')', '*', '3', '*', '2']
3. Klammern: [['5', '+', '6'], '*', '3', '*', '2']

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$
2. Tokenliste: ['(', '5', '+', '6', ')', '*', '3', '*', '2']
3. Klammern: [['5', '+', '6'], '*', '3', '*', '2']
4. Mult-Op1: [['op*', ['5', '+', '6'], '3'], '*', '2']

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$
2. Tokenliste: ['(', '5', '+', '6', ')', '*', '3', '*', '2']
3. Klammern: [['5', '+', '6'], '*', '3', '*', '2']
4. Mult-Op1: [['op*', ['5', '+', '6'], '3'], '*', '2']
5. Mult-Op2: [['op*', ['op*', ['5', '+', '6'], '3'], '2']]

Am Beispiel

Um zwischen einem schon bearbeiteten Operator und einem unbearbeiteten Operator aus der Tokenliste zu unterscheiden, bekommen die bearbeiteten den Präfix op.

1. Eingabe: $(5 + 6) * 3 * 2$
2. Tokenliste: ['(', '5', '+', '6', ')', '*', '3', '*', '2']
3. Klammern: [['5', '+', '6'], '*', '3', '*', '2']
4. Mult-Op1: [['op*', ['5', '+', '6'], '3'], '*', '2']
5. Mult-Op2: [['op*', ['op*', ['5', '+', '6'], '3'], '2']]
6. Add-Op: [['op*', ['op*', [['op+', '5', '6']] , '3'], '2']]

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.
- Wann Tupel, wann Listen?

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.
- Wann Tupel, wann Listen?
- Tupel können benutzt werden, wenn diese als Argumente übergeben und dann (neu zusammen gestellt) *zurück gegeben* werden.

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.
- Wann Tupel, wann Listen?
- Tupel können benutzt werden, wenn diese als Argumente übergeben und dann (neu zusammen gestellt) *zurück gegeben* werden.
- Da Tupel keine Seiteneffekte zulassen, ist es einfacher, damit *korrekte* Programme zu schreiben.

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.
- Wann Tupel, wann Listen?
- Tupel können benutzt werden, wenn diese als Argumente übergeben und dann (neu zusammen gestellt) *zurück gegeben* werden.
- Da Tupel keine Seiteneffekte zulassen, ist es einfacher, damit *korrekte* Programme zu schreiben.
- Sollen übergebene Listen *modifiziert* werden, muss man Listen nutzen.

Tupel versus Listen

- Das bisher Gesagte gilt sowohl für *Listen* als auch für *Tupel*.
- Wann Tupel, wann Listen?
- Tupel können benutzt werden, wenn diese als Argumente übergeben und dann (neu zusammen gestellt) *zurück gegeben* werden.
- Da Tupel keine Seiteneffekte zulassen, ist es einfacher, damit *korrekte* Programme zu schreiben.
- Sollen übergebene Listen *modifiziert* werden, muss man Listen nutzen.

→ Wir werden für den Taschenrechner *Tupel* nutzen.

Die Generierung der Tokenliste I

Tokenization

```
def tokenizer(line):
    '''Takes string and returns token list'''
    tokens = () # token list
    while (line):
        token, line = next_token(line) # get next token and re
        if token != None:
            tokens += ( token ,)
    return tokens
```

Die Generierung der Tokenliste II

Next token

```
def next_token(ln):
    '''Takes string and returns a pair (token,reststring)'''
    while ln and ln[0] == ' ':
        ln = ln[1:]
    if not ln:
        return (None, "")
    if ln[0] >= '0' and ln[0] <= '9' :
        num = 0
        while ln and ln[0] >= '0' and  ln[0] <= '9':
            num = num*10 + ord(ln[0]) - ord('0')
            ln = ln[1:]
        return (num, ln)
    else:
        return(ln[0], ln[1:])
```

Die Generierung der Tokenliste: Test

Der Tokenizer erkennt Zahlen und zerlegt alles andere in einzelne Zeichen, wobei Leerzeichen ignoriert werden.

Python-Interpreter

```
>>> tokenizer('976+6-(56*9)-ab+)')
(976, '+', 6, '-', '(', 56, '*', 9, ')', '-', 'a',
'b', '+', ')')
```

Das könnte man genereller hinbekommen!
Aber für unsere Ansprüche reicht es.

Zusammenfassung der Klammerausdrücke

Group parenthesized expressions

```
def group_paren_expr(tokens,paren):
  newtuple = ( )
  while (tokens):
    next, tokens = tokens[0], tokens[1:]
    if (next == '('):
      newel, tokens = group_paren_expr(tokens,True)
      newtuple += ( newel,)
    elif (next == ')'):
      if not paren: return (None,) , ()
      else: return newtuple, tokens
    else:
      newtuple += ( next, )
  if paren: return (None,) , ()
  else: return  newtuple , ()
```

Klammerausdrücke: Test

Python-Interpreter

```
>>> testline = "5*((12+13)*26)*9+122"
>>> tl = tokenizer(testline); tl
(5, '*', '(', '(', 12, '+', 13, ')', '*', 26, ')',
'*', 9, '+', 122)
>>> gl = group_paren_expr(tl); gl
((5, '*', ((12, '+', 13), '*', 26), '*', 9, '+',
122), ())
>>> gl[0]
(5, '*', ((12, '+', 13), '*', 26), '*', 9, '+',
122)
>>> group_paren_expr(tl)[0]
(5, '*', ((12, '+', 13), '*', 26), '*', 9, '+',
122)
```

Zwei Funktionen, die wir benötigen ...

- Die Funktion `len` liefert die Länge einer Sequenz. Kann wichtig sein, um Indexfehler zu vermeiden.

Python-Interpreter

```
>>> len(())
0
>>> len('Hello')
5
```

- Mit der Funktion `isinstance` kann man den Typ eines Objekts prüfen.

Python-Interpreter

```
>>> isinstance(5,int)
True
>>> isinstance( [ (5,), 'end'],list)
True
```

Finden von Operatoren in der Tokenliste

Find operator

```
def find_opa(opchars,tokens):
    for i,t in enumerate(tokens):
        for c in opchars:
            if c == t and i>0 and len(tokens)>i+1:
                return i
    return None
```

- Ignoriere Operatoren an erster und letzter Stelle! Dies kann nur ein Syntaxfehler sein!
- Beachte `enumerate` – hätte man auch mit `while`-Schleife hin bekommen.

Generiere Teilbäume

Group expression

```
def group_expr(tokens,opas):
    newtuple = ()
    for el in tokens:
        if isinstance(el,tuple):
            newtuple += (group_expr(el,opas),)
        else:
            newtuple += (el,)
    tokens = newtuple
    while find_opa(opas,tokens) != None:
        mix = find_opa(opas,tokens)
        tokens = (tokens[:mix-1] +
                  (("op" + tokens[mix], tokens[mix-1],
                    tokens[mix+1]),) + tokens[mix+2:])
    return tokens
```

Punkt- vor Strich-Rechnung: Test

Python-Interpreter

```
>>> testline = "5*((12+13)*26)*9+122"
>>> gl =
group_paren_expr(tokenizer(testline),False)[0]
>>> gl
(5, '*', ((12, '+', 13), '*', 26), '*', 9, '+',
122)
>>> ml = group_expr(gl,"*/"); ml
(('op*', ('op*', 5, (('op*', (12, '+', 13), 26),)),
9), '+', 122)
>>> al = group_expr(ml,"+-"); al
(('op+', ('op*', ('op*', 5, (('op*', (('op+', 12,
13),), 26),)), 9), 122),)
```

Rekursive Auswertung

Recursive evaluation

```
def eval_tree(tree):
    if isinstance(tree,int):
        return tree
    elif isinstance(tree,tuple):
        if len(tree) == 1:
            return(eval_tree(tree[0]))
        elif len(tree) == 3:
            return(eval_bin_expr(tree[0],
                                 eval_tree(tree[1]),
                                 eval_tree(tree[2])))
        else:
            return None
    else:
        return None
```

Anwendung des binären Operators (wie gehabt)

Evaluation of binary operator

```
def eval_bin_expr(op,op1,op2):
    if op1 == None or op2 == None:
        return None
    elif op == 'op+':
        return op1 + op2
    elif op == 'op-':
        return op1 - op2
    elif op == 'op*':
        return op1 * op2
    elif op == 'op/':
        return op1 // op2
    else:
        return None
```

Rekursive Auswertung: Test

Python-Interpreter

```
>>> al
(('op+', ('op*', ('op*', 5, (('op*', (('op+', 12,
13),), 26),)), 9), 122),)
>>> eval_tree(al)
29372
```

Rekursive Auswertung: Test

Python-Interpreter

```
>>> al
(('op+', ('op*', ('op*', 5, (('op*', (('op+', 12,
13),), 26),)), 9), 122),)
>>> eval_tree(al)
29372
>>> eval(testline)
29372
```

Rekursive Auswertung: Test

Python-Interpreter

```
>>> al
(('op+', ('op*', ('op*', 5, (('op*', (('op+', 12,
13),), 26),)), 9), 122),)
>>> eval_tree(al)
29372
>>> eval(testline)
29372
```

- Ja, die Python-Funktion `eval` wertet alle Programmtexte (und Ausdrücke) aus!
- Sie lässt mehr zu, als unser „Kunde“ will.
- Sie erzeugt merkwürdige Python-Fehlermeldungen.
- Die Vorgehensweise der Funktion ist aber ähnlich!

→ Können wir einen Python-Interpreter in Python schreiben?

Und jetzt alle zusammen ...

Composition of everything

```
def eval_line(line):
  return eval_tree(group_expr(
                     group_expr(
                       group_paren_expr(
                          tokenizer(line),
                          False)[0],
                       "*/"),
                     "+-"))
```