Teil VI

Reguläre Ausdrücke

Formale Sprachen

Syntax

- Jede Sprache (natürliche und formale) hat Regeln
- Für einen Text lässt sich feststellen, ob er zu einer Sprache gehört:
  - Deutsch: Der Student sitzt in der Vorlesung.
  - Python: `print "Hello World"`
- Eine Grammatik beschreibt, nach welchen Regeln eine Sprache syntaktisch aufgebaut ist.
- Bei einer Programmiersprache prüft der Compiler/Interpreter die Syntax

Semantik

- Auch bei grammatikalisch korrektem Aufbau kann ein Text sinnlos sein
  - Deutsch: Die Vorlesung sitzt in dem Studenten.
  - Python: `print "sin(x) = %f" % cos(x)`
- Eine Semantik-Korrektur für Programmiersprachen gibt es noch nicht

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- $X*$ ist die Kleensche Hülle der Menge X. Enthält beliebige Konkatenationen von Elementen aus der Menge
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)
- $N = V \backslash \Sigma$ sind die Nichtterminalsymbole/Variablen (<Subjekt>, <Verb>; A, B, ...)
- $P \subset (V^* \backslash \Sigma^*) x V^*$ ist die Menge der Produktionsregeln. Überführung eines Wortes/Texts R, das mindestens ein Nichtterminal enthält ($R \in V^* \backslash \Sigma^*$) in ein beliebiges Wort $Q \in V^*$

```
<Subj><Verb><Obj>  -->  Der Student<Verb><Objt>
AB --> AaB
```

- $S \in (V \backslash \Sigma)$ ist das Startsymbol

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Typ 1: kontextsensitive Grammatik

- Produktionsregeln der Form $\alpha B \gamma \rightarrow \alpha \beta \gamma$ (B Nichtterminal, griechische Buchstaben Worte aus V^*)
- Sprache wird von linear beschränkter Turingmaschine erkannt

Typ 2: kontextfreie Grammatik

- Produktionsregeln der Form $A \rightarrow \alpha$
- erkannt von Kellerautomat, Programmiersprachen sind Typ 2

Typ 3: reguläre Grammatik

- Produktionsregeln der Form $A \rightarrow a$ und $A \rightarrow aB$
- erkannt durch endliche Automaten / reguläre Ausdrücke

Reguläre Grammatik der Exponentialzahlen

- Einschränkende Annahmen:
  - Beide Vorzeichen (Anfang+Exponent) zwingend anzugeben
  - Genau eine Vor- und mindestens eine Nachkommastelle
- $G = (V, \Sigma, P, S)$
- $V = \Sigma \cup N$
- $N = S, M, N1, P, N2, E1, E2$
- $\Sigma = -, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e$
- Platzhalter: $z \in \{0 - 9\}$

Produktionsregeln (rechts Beispiel: -3.81e-11)

```
S  -> -M     S -> +M     | S -> -M
M  -> zP                 |   -> -3P
P  -> .N1                |   -> -3.N1
N1 -> zN2                |   -> -3.8N2
N2 -> zN2    N2 -> eE1   |   -> -3.81N2      -> -3.81eE1
E1 -> -E2    E1 -> +E2   |   -> -3.81e-E2
E2 -> zE2    E2 -> z     |   -> -3.81e-1E2   -> -3.81e-11
```

Reguläre Sprachen und Ausdrücke???

Die Menge der regulären Sprachen über einem Alphabet Σ und die zugehörigen regulären Ausdrücke sind rekursiv definiert:

- Die leere Sprache $\emptyset$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $\emptyset$.
- Der leere String $\{\wedge\}$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $\wedge$.
- Für jedes a in Σ, ist die einelementige Sprache $\{$ a $\}$ eine reguläre Sprache und der zugehörige reguläre Ausdruck ist a
- Wenn A und B reguläre Sprachen sind mit zugehörigen regulären Audrücken $r1$ and $r2$, dann
  - ist $A \cup B$ (Vereinigung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1|r2)$
  - ist AB (Verknüpfung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1r2)$
  - ist A^* (Kleene star) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1^*)$

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen
- Alphabet: $\{-, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$
- Einelementige Sprachen: $S_- = \{-\}$, $S_+ = \{+\}$, $S_0 = \{0\}$, $S_1 = \{1\}$, ..., $S_. = \{.\}$, $S_e = \{e\}$
- zugehörige regulären Ausdrücke: $r_- = -, r_+ = +, r_0 = 0, r_1 = 1, ..., r_. = ., r_e = e$
- Vereinigung zweier Sprachen: $\{a, b\} \cup \{0, 1, 2\} = \{a, b, 0, 1, 2\} \Rightarrow |S_1| + |S_2|$ Elemente
- Verknüpfung zweier Sprachen: $\{a, b\}\{0, 1, 2\} = \{a0, a1, a2, b0, b1, b2\} \Rightarrow |S_1| \cdot |S_2|$ Elemente
- Kleensche Hülle einer Sprache: $\{0, 1\}^* = \{\emptyset, 0, 1, 00, 01, 10, 11, 000, ...\} \Rightarrow \infty$ Elemente

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (-\|+)$
$S_0 \cup S_1 \cup$	$S_B = \{0, 1, ..., 9\}$	$r_B = (0\|1\|...\|9) = (0-9)$
$S_A S_B S.$	$S_C = \{-0., ..., -9., ...\}$	$r_C = (-\|+)(0-9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, ...\}$	$r_D = (-\|+)(0-9).(0-9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, ...\}$	$r_E = (-\|+)(0-9).(0-9)(0-9)^*$
		$r_E = (-\|+)(0-9).(0-9)^+$
$S_E S_e S_A$	$S_F = \{-0.0e, ...\}$	$r_F = (-\|+)(0-9).(0-9)^+e$
$S_F S_A$	$S_G = \{-0.0e-, ...\}$	$r_G = (-\|+)(0-9).(0-9)^+e(-\|+)$
$S_F S_B$	$S_H = \{-0.0e-0, ...\}$	$r_H = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)$
$S_H S_B^*$	$S_I = \{-0.0e-00, ...\}$	$r_I = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0-9).(0-9)^+e(-|+)(0-9)^+$ ist der zugehörige reguläre Ausdruck
- in Python: `rI = "[-+][0-9]\.[0-9]+e[-+][0-9]+"`
- noch kürzer: `rI = "[-+]\d\.\d+e[-+]\d+"`

Regular Expressions!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Einschränkungen

- Für Anfänger sehr kryptisch
- Nach reiner Lehre „zählen“nicht möglich (z.B. $a^n b^n$ geht nicht)

Vorteile

- Sehr nützliches Werkzeug zum Finden von Pattern
- Vergleichbare „manuelle“Implementierung viel aufwändiger
- Python hat Erweiterungen, die sogar „zählen“ermöglichen

Reguläre Ausdrücke in Python

Pattern Syntax

- Regular expressions should always be placed in a raw string
- E.g. `r'([^.]*.(.*))'`

`.`	entspricht beliebigem Zeichen außer newline
`^`	entspricht dem Beginn eines strings
`$`	entspricht dem Ende eines strings
`*`	voriger Ausdruck beliebig oft (incl. Null mal) (gierig)
`+`	voriger Ausdruck beliebig oft (excl. Null mal) (gierig)
`?`	voriger Ausdruck Null- oder einmal (gierig)
`*?, +?, ??`	nicht gieriger Versions von `*, +, ?`
`{m}`	voriger Ausdruck genau m Mal
`{m,n}`	voriger Ausdruck m bis n Mal (gierig)
`{m,n}?`	nicht gierige Version von `{m,n}`
`[...]`	ein beliebiges Zeichen aus der Menge (...)
`[^...]`	ein beliebiges Zeichen, das nicht in der Menge ist
`A|B`	A oder B (A und B sind reguläre Ausdrücke)
`(...)`	Speichert den gefundenen Inhalt der Klammern

Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines \verwendet werden, z.B. `\.`, `\*`
- Normale Maskierungszeichen (`\n`, `\t`, `...`) funktionieren wie erwartet, z.B. `r'\n+'` entspricht einer oder mehreren Zeilenumbrüchen

`\`*number*	enspricht dem n-ten zuvor gefundenen Text (starting from 1)
`\d`	entspricht `[0-9]`
`\D`	entspricht `[^0-9]`
`\s`	enspricht beliebigem whitespace (= `[\t\n\r\f\v]`)
`\S`	alles außer whitespace
`\w`	beliebiges alphanumerisches Zeichen
`\W`	beliebiges nicht-alphanumerisches Zeichen
`\A`	entspricht dem Beginn eines strings
`\Z`	entspricht dem Ende eines strings

Regular Expressions - erste Python-Beispiele

- `findall(patt, string)` – finde alle nicht überlappenden Vorkommen
- `sub(patt, repl, string)` – ersetze Vorkommen durch `repl`

```
regstr = r'\d*\.\d*|\d*'
s = "12.3/5/4.4;5.7;6"
findall(regstr,s)
regstr = r'(\d*\.\d*|\d*)'
sub(regstr, r'\1xxx', s)
```

Match-Objekte

- Manche Funktionen aus `re` geben keine strings, sondern „Match-objekte“(m) zurück
- `match(patt, string)` Sucht nach Übereinstimmung am Beginn des strings
- `search(patt, string)` Sucht die erste Übereinstimmung im string
- `finditer(patt, string)` wie findall, gibt aber einen iterator zurück
- `m.group(i)` Gibt Übereinstimmungsgruppe i zurück ($i \geq 1$)
- `m.groups()` Gibt Tupel mit allen Übereinstimmungsgruppen zurück

```
import re
s = "Ferien␣12/24/2010␣-␣01/06/2011"
patt = r'(\d+)/(\d+)/(\d+)'
for m in re.finditer(patt,s):
  print("%s.%s.%s" % (m.group(2), m.group(1), m.group(3)))
```

Teil VII

Exceptions

Exceptions: Fehler abfangen

- Fehler treten beispielsweise auf, wenn ein Benutzer interagiert:

```
x = raw_input("Bitte␣eine␣Zahl␣eingeben:␣")
print float(x)
```

- Wenn die Konvertierung nach `float` nicht möglich ist, wird eine `ValueError` exception geworfen

```
Please enter a number: xyz
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): xyz
```

- Um mit diesem fehlerhaften Verhalten umzugehen, kann die exception abgefangen und bearbeitet werden

Syntax für Exceptions

- `try`: leitet einen Block ein, der normal ausgeführt wird
- Darauf folgen (mehrere) `except`-Anweisungen, die im ensprechenden Fehlerfall ausgeführt werden
- Dann optional ein `else`, dass nur ausgeführt wird, wenn kein Fehler auftritt
- zuletzt optional `finally`, das immer ausgeführt wird

```
try:
    # beliebiger Code
except ValueError as e: # Py>3.0: "Exception as e"
    # behandle ValueError
except Exception as e:
    # behandle alle übrigen Fehler
else:
    # ausführen, falls keine Fehler auftrat
finally:
    # Wird immer ausgeführt
```

Beispiel

Wieder das Beispiel mit User-Eingabe

```
eingabe = False
while not eingabe:
    x = raw_input("Bitte eine Zahl eingeben: ")
    try:
        print float(x)
        eingabe = True
    except ValueError as e:
        print "Das war keine Zahl!"
        # evtl. noch irgendwas mit e machen
```

- Eigene Exceptions werfen (Hierfür können Standardfehler verwendet werden, oder eigene Fehlertypen erzeugt werden):

```
raise RuntimeError("Ooops! Irgendwas ging schief!")
raise IOError("Datei existiert nicht...")
```

Eingebaute Exceptions

```
BaseException                 # Wurzel aller exc.
  GeneratorExit
  KeyboardInterrupt           # z.B. Ctrl+C
  SystemExit                  # Program Exit (sys.exit())
  Exception                   # Basis für nicht-exit exc.
    StopIteration
    StandardError             # Nur Python 2.x
      ArithmeticError
        FloatingPointError
        ZeroDevisionError
      AssertionError
      AttributeError
      EnvironmentError
        IOError               # z.B. bei open("datei.txt")
        OSError
      EOFError
      ImportError             # z.B. Modul nicht gefunden
```

```
LookupError
  IndexError          # z.B. bei Tupeln/Listen
  KeyError            # z.B. bei Dictionaries
MemoryError
NameError             # Name nicht gefunden
UnboundedLocalError
ReferenceError
RuntimeError
NotImplementedError
SyntaxError
  IndentationError
    TabError
SystemError
TypeError             # Typ-Fehler (2 + "3")
ValueError            # Wert-Fehler (float("drei"))
  UnicodeError
    UnicodeDecodeError
    UnicodeEncodeError
    UnicodeTranslateError
```

Build-In Exception werfen

```
def readfloat1():
  while True:
    try:
      a = raw_input("Zahl␣zwischen␣0.0␣und␣1.0:␣")
      a = float(a)
      if(a<0.0 or a>1.0):
        raise ValueError("Zahl␣nicht␣in␣[0.0;␣1.0]")
    except ValueError as e:
      print "Fehler:␣%s" % e
    else:
      return a
```

- Bei `a = float(a)` wird automatisch Fehler geworfen, der Rest des try-Blocks wird nicht mehr ausgeführt
- Falls `a` konvertierbar ist, wird der Wertebereich überprüft und ggf neue Exception geworfen

Verschachtelte Try-Blöcke

```
def readfloat2():
  while True:
    try:
      a = raw_input("Zahl zwischen 0.0 und 1.0: ")
      try:
        a = float(a)
      except ValueError as e:
        raise ValueError("Das ist keine Zahl: %s" % a)
      if(a<0.0 or a>1.0):
        raise ValueError("Zahl nicht in [0.0; 1.0]")
    except ValueError as e:
      print "Fehler: %s" % e
    else:
      return a
```

- Exceptions können abgefangen und gleich wieder geworfen werden

Eigene Exceptions

- Hierfür sind Klassen nötig, die wir erst in der nächsten Vorlesung behandeln
- Trotzdem schon mal ein kleiner Ausblick:

```
class MeinFehler(Exception): pass

raise MeinFehler("schlimmer␣Fehler")
```

- Man kann natürlich auch kompiziertere Dinge tun:

```
class KomplizierterFehler(Exception):
  def __init__(self, nummer, nachricht):
    self.args = (nummer, nachricht)
    self.nummer = nummer
    self.nachricht = nachricht
  def machIrgendwas():
    ...
```