easyparser

Easyparser ist ein Python-Package, das es erlaubt, eine Grammatik in Python zu definieren und anschliessend daraus einen Recursive Descent Parser zu generieren, mit dem Sätze aus dieser Grammatik geparst werden können.

Der erzeugte Parser cacht bereits erzeugte Teile eines Parse-Baums und ist deshalb für einfachere Grammatiken mässig effizient. Er kommt auch mit mehrdeutigen Grammatiken klar, d.h. er findet sämtliche der Grammatik entsprechenden Parsebäume für mehrdeutige Sätze, nicht nur einen. Ausserdem ist eine Erweiterung zu Feature-basierteren Grammatiken geplant.

Dieses Package eignet sich deshalb besonders zum raschen Experimentieren mit kleinen Grammatiken für natürliche Sprache.

Grammatiken definieren

Jeder Parser benötigt eine Grammatik, deren Sätze (bzw. Worte) er erkennen soll. Fangen wir mit einem einfachen Beispiel an: Wir wollen Sätze wie "Der Hund bellt" oder "Das Krokodil schwimmt" erkennen und jeweils das involvierte Tier sowie die von ihm ausgeführte Handlung erkennen:

S → NP V
NP → ART NN
ART → `Der` | `Die` | `Das`
NN → `Hund` | `Schnecke` | `Krokodil`
V → `bellt` | `schleicht` | `schwimmt`

Diese Grammatik können wir wie folgt mit easyparser umsetzen:

```
from easyparser import *

s = Nonterminal('Satz')
np = Nonterminal('Nominalphrase')
nn = Nonterminal('Nomen')
v = Nonterminal('Verb')
art = Nonterminal('Artikel')

s >> np + v
np >> art + nn
art >> Terminal('Der') | Terminal('Die') | Terminal('Das')
nn >> Terminal('Hund') | Terminal('Schnecke') | Terminal('Krokodil')
v >> Terminal('bellt') | Terminal('schwimmt' | Terminal('schleicht')
```

Das ist bereits eine funktionsfähige Grammatik. Folgendermassen erzeugen wir einen Parser dafür:

```
parser = RecursiveDescentParser(s)
```

Wir müssen dem Parser nun einen Eingabestrom mit Terminals vorsetzen, für die er dann einen Parse-Baum erzeugt. Das können wir z.B. wie folgt erledigen:

```
eingabe = raw_input('Gib was ein: ')
terminals = eingabe.split(' ')
for semantic in parser.parse(terminals):
   print 'Aha, ich verstehe: ', semantic
print 'Keine weiteren Parses.'
```

`terminals` enthält nach dem `split` eine Liste von einzelnen Worten; das ist natürlich recht anfällig, da wir uns nicht darum kümmern, Satzzeichen und so weiter zu entfernen, sondern einen Satz einfach bei Leerzeichen in Terminals zerlegen. Die `for`-Schleife gibt nacheinander sämtliche gefundenen Parsebäume für den eingetippten Satz aus. In diesem Fall wird das maximal einer sein, da die Grammatik nicht mehrdeutig ist:

```
Gib was ein: Das Krokodil schleicht
Aha, ich verstehe:  ('Satz', [('Nominalphrase',
[('Artikel', ['Das']), ('Nomen', ['Krokodil'])]),
 ('Verb', ['schleicht'])])
Keine weiteren Parses.
```

Auf den ersten Blick sieht das erst einmal recht verwirrend aus. Beim genaueren Hinsehen sieht man aber, dass der Parser den Satz erfolgreich auf das Satzsymbol `Satz` reduzieren konnte: Der eingegebene Satz bestehend aus einer `Nominalphrase` und einem `Verb` (`'schleicht'`); die `Nominalphrase` besteht dabei aus einem `Artikel` (`'Das'`) und einem `Nomen` (`'Krokodil'`).

Warum sieht die Ausgabe des Parsers so kompliziert aus? Das hat hauptsächlich damit zu tun, dass der Parser, wenn wir ihm keine genaueren Anweisungen

geben, alle erkannten Nonterminals in ein Tupel packt, die als erstes Element den Namen des Nonterminals und als zweites Element eine Liste der Elemente enthalten, welche dieses Nonterminal ausmachen. Damit lässt sich die gesamte grammatische Struktur des erkannten Satzes abbilden.

Parse-Aktionen

Da wir ja nicht an der gesamten Satzstruktur, sondern nur am Nomen und am Verb interessiert sind, können wir den Parsebaum, den der Parser uns zurückliefert und den wir im folgenden als *semantischen Wert* des geparsten Satzes bezeichnen werden, vereinfachen. Das müssen wir dem Parser mit Parse-Aktionen beibringen. Die Hauptaufgabe einer Parse-Aktion ist es, einen semantischen Wert für ein Nonterminal zu erzeugen.

Wenn der Parser beispielsweise eine Nominalphrase erkennt, kann er den Artikel verwerfen, da wir nicht daran interessiert sind. Wir legen das mit einer Parse-Aktion fest, die immer dann aufgerufen wird, wenn der Parser eine Nominalphrase erkennt:

```
def np_action(artikel, nomen):
   return nomen

parser.setParseAction(np, np_action)
```

Eine Parse-Aktion eines Nonterminals erhält als Argumente die semantischen Werte der Elemente, die zu diesem Nonterminal reduziert wurden. In unserem Fall werden ein Artikel und ein Nomen zu einer Nominalphrase reduziert, also erhält die Parse-Aktion vom Parser den semantischen Wert des Artikels und des Nomens. Mit der obigen Funktion erreichen wir, dass der semantische Wert des Nomens auch als semantischer Wert der Nominalphrase verwendet wird. Den Artikel verwerfen wir.

Aber was ist der semantische Wert eines Nomens?

Nun, da wir dem Parser nichts anderes dazu gesagt haben, ist dies (für den Fall, dass wir immer noch den Satz 'Das Krokodil schleicht' analysieren) `('Nomen', ['Krokodil'])`. Das scheint unnötig kompliziert, aber wir können dem Parser helfen, das zu vereinfachen:

```
def nn_action(nomen):
   return nomen

parser.setParseAction(nn, nn_action)
```

Das Nonterminal `nn` besteht aus einem von drei möglichen Terminals; der semantische Wert, den die Parse-Aktion für `nn` übergeben bekommt, ist also `'Krokodil'` (oder eines der beiden anderen Nomen). `nn_action` sorgt dafür, dass dieser String auch als semantischer Wert des Nonterminals `nn` fungiert.

Der Rückgabewert des Parsers sieht jetzt schon viel einfacher aus:

```
Gib was ein: Das Krokodil schleicht
Aha, ich verstehe:  ('Satz', ['Krokodil', ('Verb', ['schleicht'])])
Keine weiteren Parses.
```

Wenn wir uns nun noch darum kümmern, das Verb- und das Satz-Nonterminal ebenfalls zu vereinfachen, erreichen wir die gewünschte einfache Darstellung:

```
def v_action(verb):
   return verb

def s_action(np, verb):
   return [np, verb]

parser.setParseAction(v, v_action)
parser.setParseAction(s, s_action)
```

Interessant ist hier vor allem die Parse-Aktion für das Satz-Nonterminal: wir fassen den semantischen Wert der Nominalphrase und den semantischen Wert des Verbs in einer einfachen Liste zusammen. Der semantische Wert eines Satzes ist jetzt also eine einfache Liste mit zwei Elementen:

```
Gib was ein: Das Krokodil schleicht
Aha, ich verstehe:  ['Krokodil', 'schleicht']
Keine weiteren Parses.
```

Natürlich hätte dieses Resultat auch ohne Parser realisiert werden können. Die Grammatik ist so einfach, dass man sich mit Python-Bordmitteln hätte behelfen können. Das hier gezeigte Vorgehen bewährt sich aber auch für umfangreichere Grammatiken.

Parse-Aktionen für Alternativen

Wir haben im letzten Beispiel eine sehr einfache Grammatik verwendet. Wie gehen wir mit dem folgenden, etwas erweiterten Definition einer Nominalphrase um?

```
adj = Nonterminal('Adjektiv')

np >> art + adj + nn | art + nn
```

Eine Nominalphrase besteht nun aus entweder 2 oder 3 Elementen. Wie definieren wir dafür eine Parse-Aktion?

```
def np_action(art, adj, nn):
   ...

def np_action(art, nn):
   ...
```

Obige zweifache Definition von `np_action` funktioniert nicht; Python wird die erste Definition einfach mit der zweiten überschreiben. Für dieses Problem gibt es zwei Lösungen. Die erste wird oft ausreichen:

```
def np_action(*args):
   if len(args) == 3:
      art, adj, nn = args
      ...
   elif len(args) == 2:
      art, nn = args
      ...
   else:
      # Kann nicht vorkommen
      assert False
```

Diese Lösung funktioniert, wenn nur aufgrund der Anzahl Elemente unterschieden werden kann, welche Alternative der rechten Seite des Nonterminals wir vor uns haben. Sie versagt jedoch beispielsweise in folgendem Fall:

```
quant = Nonterminal("Quantifizierung")

np >> art + nn | quant + nn
quant >> Terminal('Viele') | Terminal('Einige')
```

Hier ist nun nicht mehr nur aufgrund der Anzahl Elemente ersichtlich, ob es sich bei den semantischen Werten, die der Parse-Aktion für `np` übergeben werden, um die semantischen Werte für einen Artikel und ein Nomen oder um die Werte für einen Quantifizierer und ein Nomen handelt. Das ist vielleicht nicht ein Problem (nämlich wenn die semantischen Werte einfach 'durchgereicht' werden); wenn aber basierend auf ihnen ein komplizierterer semantischer Wert für `np` aufgebaut werden soll, müssen wir die Grammatik selbst umschreiben:

```
np1 = Nonterminal("Nominalphrase 1")
np2 = Nonterminal("Nominalphrase 2")
quant = Nonterminal("Quantifizierung")

np >> np1 | np2
np1 >> art + nn
np2 >> quant + nn
```

In dieser Grammatik können wir nun für das Nonterminal `np1` und das Nonterminal `np2` separate Parse-Aktionen festlegen und die Parse-Aktion von `np` zu einem einfachen "Durchreicher" degradieren.

Linksrekursive Grammatiken

Linksrekursive Grammatiken stellen für Recursive Descent Parser ein Problem dar, da sie unweigerlich in eine endlose Rekursion führen:

```
s = Nonterminal('Satz')

s >> s + Terminal('und') + s
...
```

Diese Grammatik ist linksrekursiv (weil das Nonterminal `s` auf der rechten Seite einer Definition von `s` an erster Stelle steht) und der Parser wird sich in ihr verlieren. Diese Grammatik muss von Hand in eine Grammatik überführt werden, welche nicht mehr linksrekursiv ist, damit sie geparst werden kann.

Parse-Aktionen mit Epsilon oder None

Grammatiken können auch Nonterminals enthalten, welche zu gar nichts expandieren, bzw. welche aus dem Nichts spontan generiert werden können:

```
np >> art + adj + nn
adj >> Terminal('schöne') | None
```

Hier kann ein für die Reduktion einer Nominalphrase `np` benötigtes Nonterminal `adj` auch spontan generiert werden, wenn 'schöne' im Eingabestrom nicht vorhanden ist, da für adj auch die Alternative `None`, d.h. nichts, angegeben wurde.[1]

Die Alternative None wird wie ein vollwertiges Element der Grammatik behandelt; in Parse-Aktionen erhalten also auch für die Alternative `None` einen semantischen Wert. Dieser ist `None`.

Der Eingabestrom: Tokens

Easyparser parst einen Strom von *Tokens*. Es gibt einen Unterschied zwischen einem Token und einem Terminal, der aber bisher nicht wichtig war, weil Tokens, Terminals und ihr semantischer Wert bisher identisch waren. Um den Satz "Das Krokodil rennt" zu parsen, musste der Satz in die einzelnen Worte "Das", "Krokodil" und "rennt" zerlegt werden. Diese drei Worte waren der Eingabestrom, den der Parser verarbeitete.

Da diese Worte in der Grammatik als Terminals definiert waren und der semantische Wert die Worte selbst waren, gab es keinen Grund, zwischen Tokens und Terminals zu unterscheiden: Das Token "Krokodil" war identisch mit dem Terminal "Krokodil" und sogar mit dem semantischen Wert dafür (wieder "Krokodil").

Diese Identität ist aber keineswegs zwingend. Aus jedem Token im Eingabestrom muss sich ein eindeutiges Terminal und ein eindeutiger semantischer Wert extrahieren lassen, aber die drei Elemente Token, Terminal und semantischer Wert müssen nicht identisch sein.

[1] Achtung: Zur Zeit hat der Parser Probleme mit Grammatiken, welche `None` enthalten: Es werden zuviele mögliche alternative Parsebäume gefunden, was im schlimmsten Fall dazu führen kann, dass der Parser unendlich viele Parsebäume findet. Wird *None* in einer Grammatik verwendet, sollte man sich momentan mit dem ersten gefundenen Parsebaum begnügen.

Warum dies wichtig ist

Gerade wenn wir mit natürlicher Sprache arbeiten, ist es schwierig, alle Wörter (Terminals), die in Sätzen auftauchen können, vorgängig in der Grammatik als Terminals zu verankern. Die Grammatik wird dadurch riesig und unübersichtlich. Besser ist es, die Grammatik mit Wortklassen aufzubauen statt mit einzelnen Wörtern:

S → NP V
NP → ART NN

In dieser Grammatik sind ART, NN und V Terminals. Die eigentlichen Worte ("Der", "Das", "Krokodil", "Schnecke", "rennt" usw.) kommen in der Grammatik nicht mehr explizit vor; sie bilden aber die semantischen Werte der Terminals ART, NN und V. Wir lösen die Grammatik und damit den Parser also von der Wortebene und heben ihn auf die Ebene von Wortklassen.

Natürlich benötigen wir nun einen zusätzlichen Schritt, der einen Satz wie "Das Krokodil rennt" in die Terminals ART, N und V übersetzt, bevor wir diese Terminals als Eingabestrom dem Parser übergeben können.

Dieser Schritt war vorher nicht notwendig, weil die einzelnen Worte "Das", "Krokodil" und "rennt" des Satzes bereits Terminals aus der Grammatik repräsentierten; wir konnten sie direkt als Eingabestrom dem Parser übergeben.

Es ergibt sich auch noch ein zweites Problem: Wenn wir dem Parser für den Satz "Das Krokodil rennt" nur den Eingabestrom ART, NN und V übergeben würden, dann wäre der Parser nicht mehr in der Lage, irgendetwas über die Semantik (die Bedeutung) des Satzes auszusagen, da er über den eigentlichen Inhalt des Satzes, die Worte, nicht mehr verfügen würde.

Wir müssen dem Parser also nun jeweils Paare übergeben, die aus der Wortklasse und dem eigentlichen Wort bestehen, z.B. so:

```
[ ('ART', 'Das'), ('NN', 'Krokodil'), ('V', 'rennt') ]
```

Entsprechend benötigen wir zusätzlich zum Parser einen Programmteil, der aus einem gewöhnlichen Satz diese Liste von Paaren bildet. Jedes solche Paar repräsentiert anschliessend ein Token des Eingabestroms für den Parser.

Änderungen am Quellcode

Natürlich müssen wir dem Parser nun auch mitteilen, wie er aus einem Token das Terminal und den dazugehörenden semantischen Wert extrahieren kann. Die tun wir am einfachsten mit zwei kleinen Funktionen:

```
def extractTerminal(token):
   return token[0]

def extractSemantic(token):
   return token[1]
```

Bei der Konstruktion des Parsers müssen wir nun neben dem Satzsymbol der zu parsenden Grammatik auch die beiden Extraktionsfunktionen angeben:

```
parser = RecursiveDescentParser(s, extractTerminal, extractSemantic)
```

Damit ist es nun möglich, einen Eingabestrom wie den obigen zu parsen:

```
tokens = [ ('ART', 'Das'), ('NN', 'Krokodil'), ('V', 'rennt') ]
for semantic in parser.parse(tokens):
   print 'Aha, ich verstehe: ', semantic
print 'Keine weiteren Parses.'
```

Was wurde dadurch erreicht? Der Parser kann nun sämtliche Sätze der Form *Artikel, Nomen, Verb* parsen. Die Schwierigkeit, einzelne Wörter einer Wortklasse zuzuweisen, verschwindet natürlich nicht, sondern verschiebt sich nur auf die Phase, in welcher der Token-Eingabestrom erzeugt werden muss, aber die Parser-Komponente und die Grammatik vereinfachen sich damit massiv.

Komponentenbauweise mit Scanner

Wir haben es hier mit einer in der Softwareentwicklung typischen Aufteilung von Arbeit auf mehrere Schichten oder Komponenten zu tun.

Easyparser erleichtert uns anfänglich die Arbeit, indem es automatisch davon ausgeht, dass die übergebenen Tokens gleichzeitig auch Terminals und ihre semantischen Werte repräsentieren.

Die Möglichkeit, die Extraktion von Terminals und semantischen Werten aus Tokens selbst zu steuern, ermöglicht es aber, den Parser auch für anspruchsvollere Aufgaben zu verwenden.

Die Aufteilung des gesamten Analysevorgangs auf mehrere Phasen ist übrigens nicht nur in Bezug auf natürliche Sprache nützlich; auch Parser für formalere Sprachen wie Programmiersprachen teilen den Parse-Vorgang klassischerweise in eine lexikalische Analyse (Lexer oder Scanner) und die eigentliche grammatische Analyse auf.

Die Aufteilung in Scanning- und Parsing-Phase vereinfacht beide Komponenten. Die Scanning-Phase ist grundsätzlich damit beschäftigt, einen Eingabestrom aus einzelnen Zeichen in grössere Sinneinheiten, die Tokens, zu gruppieren, und diese Tokens zu klassifizieren.

Im Fall der Analyse von Programmquellcode wird der Scanner verschiedene Token-Klassen wie Befehlsworte, Variablennamen, Operatoren, Ganzzahlen etc. identifizieren. Das ist meist bereits mit einer Typ 3 - Grammatik (auf der Chomsky-Skala) problemlos möglich; praktisch immer reicht es sogar aus, das erste Zeichen eines Tokens anzuschauen, um herauszufinden, zu welcher Klasse ein Token gehört.[2]

[2]Das ist natürlich kein Zufall. Dadurch wird ermöglicht, dass effiziente Scanner für Programmiersprachen problemlos von Hand geschrieben werden können, was früher wichtig war.

Im Fall von natürlicher Sprache ist die Gruppierung von einzelnen Zeichen zu Worten und deren Einordnung in Wortklassen schwieriger, funktioniert aber mit *statistischen Taggern* bereits recht zuverlässig.

Installation

Easyparser ist ein Python-Package, das aus mehreren Quelldateien zusammengesetzt ist (`grammar.py`, `recursivedescent.py` und `features.py`, sowie die Datei `__init__.py` für die Paket-Initialisierung). Diese sind alle in einem Ordner namens `easyparser` abgelegt.

Verwendung ohne Installation

Die einfachste Möglichkeit, easyparser zu verwenden, ist, den `easyparser`-Ordner, der diese Quelldateien enthält, im gleichen Ordner abzulegen, in dem auch die Python-Datei liegt, welche easyparser verwenden möchte. Easyparser lässt sich dann mit einem einfachen import-Befehl verwenden:

```
from easyparser import *
```

Dies ist dann empfehlenswert, wenn der Benutzer keine Administratorrechte hat, um Python-Pakete zu installieren, oder wenn easyparser nur gerade in einem einzigen Projekt verwendet werden soll.

Installation in Standard-Packages-Ordner

Der easyparser-Ordner, der die oben genannten Python-Quelldateien enthält, kann im Standard-Packages-Ordner der lokalen Python-Installation abgelegt werden. In dem Fall sollte easyparser anschliessend allen Benutzern ohne weitere Arbeit zur Verfügung stehen.

Installation in Nicht-Standard-Ordner

Das easyparser-Package kann auch in einem Ordner abgelegt werden, den Python nicht standardmässig nach Paketen durchsucht. In dem Fall muss aber vor dem import des easyparser-Packages der Ordner in den Suchpfad aufgenommen werden:

```
import sys
sys.path.append("/opt/packages/")

from easyparser import *
```

Im Beispiel oben wird der Pfad zu `opt/packages` manuell hinzugefügt; der easyparser-Ordner mit den Python-Quelldateien sollte direkt unter diesem Ordner liegen.

Ein "richtiges" Beispiel: SQL parsen

Die bisher vorgestellten Parser waren trivial. Ein brauchbares Beispiel mit einer formalen Sprache gefällig? Stellen wir uns vor, wir haben in Python einige Listen mit Daten, z.B. die folgenden:

```
personen = [['id', 'vorname', 'nachname', 'alter'],
            [1, 'Bart', 'Simpson', 10],
            [2, 'Lisa', 'Simpson', 8],
            [3, 'Maggie', 'Simpson', 1]]
spielzeug = [['name', 'besitzer'],
             ['Saxophon', 2],
             ['Krusty-Puppe', 1],
             ['Steinschleuder', 1],
             ['Schnuller', 3]]
```

Es wäre hübsch, wenn wir Daten aus diesen Tabellen mit einer SQL-ähnlichen Sprache abfragen könnten. Erstellen wir zuerst einmal die Grammatik für die Abfragesprache:

S → `select` SPALTEN `from` TABELLEN `where` BEDINGUNG
SPALTEN → `name` | `name ,` SPALTEN
TABELLEN → `name` | `name ,` TABELLEN
BEDINGUNG → `name = name`
BEDINGUNG → `name = string`
BEDINGUNG → `name = number`

Diese Grammatik soll einfache Abfragen wie beispielsweise `SELECT alter FROM personen WHERE vorname = 'Bart'` erlauben. Wir verwenden in der Grammatik einige Terminals, welche ein Scanner für den Parser übersetzen soll. Insbesondere stehen die Terminals `select`, `from` und `where` für die Worte `SELECT`, `FROM` und `WHERE`, sowie `,` und `=` stehen für die entsprechenden Literale. Ausserdem haben wir die Terminals `name`, `string` und `number`, die etwas komplizierter sind: `string` steht für den Inhalt eines mit einfachen Anführungszeichen umgebenen Strings, `number` für eine beliebige Ganzzahl und `name` für ein einfaches Wort. Wir werden uns zuerst um diese Übersetzung kümmern:

```
def literal2Token(literal):
   if literal in ['SELECT', 'FROM', 'WHERE', ',', '=']:
      return (literal.lower(), literal)
   elif literal[0] == "'" and literal[-1] == "'":
      return ('string', literal[1:-1])
   elif literal[0] in ['1', '2', '3', '4', '5', '6', '7', '8', '9']:
      return ('number', int(literal))
   else:
      return ('name', literal)
```

Nun schreiben wir etwas Code, um diese Routine zu testen:

```
eingabe = raw_input("Abfrage: ")
literals = eingabe.split(" ")
for literal in literals:
   print literal2Token(literal)
```

Und nun der Test (*demo/sql1.py*):

```
Abfrage: SELECT alter FROM personen WHERE vorname = 'Bart'
('select', 'SELECT')
('name', 'alter')
('from', 'FROM')
('name', 'personen')
('where', 'WHERE')
('name', 'vorname')
('=', '=')
('string', 'Bart')
```

Das funktioniert schon einmal prächtig. Wir können nun also die Terminals, welche unsere Grammatik benötigt, generieren. Tatsächlich generieren wir im obigen Beispiel nicht nur Terminals, sondern Tokens; das erste Element jedes Token-Tupels ist das Terminal und das zweite Element stellt jeweils den semantischen Wert dar.

Als nächstes kümmern wir uns um die Definition der Grammatik mit Easyparser:

```
s = Nonterminal('SQL')
spalten = Nonterminal('Spalten')
tabellen = Nonterminal('Tabellen')
bedingung = Nonterminal('Bedingung')
T = Terminal

s >> T('select') + spalten + T('from') + tabellen + T('where') + bedingung
spalten  >> T('name') | T('name') + T(',') + spalten
tabellen >> T('name') | T('name') + T(',') + tabellen
bedingung >> T('name') + T('=') + T('string')
bedingung >> T('name') + T('=') + T('number')
bedingung >> T('name') + T('=') + T('name')
```

Den Parser generieren wir wie folgt:

```
def extractTerminal(token):
   return token[0]

def extractSemantic(token):
   return token[1]

parser = RecursiveDescentParser(s, extractTerminal, extractSemantic)
```

Anschliessend können wir mit folgendem Code Abfragen an den Parser übergeben:

```
eingabe = raw_input("Abfrage: ")
literals = eingabe.split(" ")
tokens = [ literal2Token(literal) for literal in literals ]
print "Tokens: ", tokens
for semantic in parser.parse(tokens):
   print 'Aha, ich verstehe: ', semantic
print 'Keine weiteren Parses.'
```

Dass dies funktioniert, zeigt ein Versuch (*demo/sql2.py*):

```
Abfrage: SELECT alter FROM personen WHERE vorname = 'Bart'
Tokens:  [('select', 'SELECT'), ('name', 'alter'),
('from', 'FROM'), ('name', 'personen'), ('where', 'WHERE'),
 ('name', 'vorname'), ('=', '='), ('string', 'Bart')]
Aha, ich verstehe:  ('SQL', ['SELECT', ('Spalten', ['alter']),
'FROM', ('Tabellen', ['personen']), 'WHERE', ('Bedingung',
['vorname', '=', 'Bart'])])
Keine weiteren Parses.
```

Nun müssen wir dafür sorgen, dass der Parser tatsächlich etwas sinnvolles tut, statt nur den Parsebaum auszugeben. Wir definieren also Parse-Aktionen, welche die Abfrage auf den Daten tatsächlich durchführen.

Eigentlich ist ja recht klar, was wir tun müssen: Wir müssen uns die Tabelle, die der Parser im TABELLEN-Nonterminal erkennt, anschauen, sämtliche Spalten, die im SPALTEN-Nonterminal nicht erwähnt sind, wegwerfen, und aus der übriggebliebenen Tabelle diejenigen Zeilen wegwerfen, die nicht der angegebenen BEDINGUNG entsprechen. Das, was übrig bleibt, ist das Resultat unserer Abfrage.

Wir benötigen als erstes ein Verzeichnis sämtlicher Tabellen, die uns zur Verfügung stehen. Dies realisieren wir wie folgt:

```
alletabellen = { 'personen': personen,
             'spielzeug': spielzeug}
```

Nun sorgen wir dafür, dass der semantische Wert des Nonterminals `TABELLEN` diejenige Python-Liste ist, die in der Abfrage ausgewählt wurde:

```
def tabellen_action(*args):
   global alletabellen
   if len(args) == 1:
      return tabellen[args[0]]
   else:
      raise ValueError("Joins noch nicht eingebaut")
```

Wir beschränken uns im Moment auf eine einzige Tabelle, SQL-Joins sind also noch nicht möglich, obwohl die Grammatik sie erlauben würde.

Als nächstes kümmern wir uns darum, die Liste der in der Abfrage angegebenen Spalten zu bauen:

```
def spalten_action(*args):
   if len(args) == 1:
      return [args[0]]
   elif len(args) == 3:
      return args[2].insert(args[0])
   else:
      assert False
```

Wenn wir diese Parse-Aktionen im Parser installieren, können wir das Zwischenresultat bereits begutachten (*demo/sql3.py*):

```
Abfrage: SELECT vorname , alter FROM personen WHERE vorname = 'Bart'
Aha, ich verstehe:  ('SQL', ['SELECT', ['vorname', 'alter'], 'FROM',
[['id', 'vorname', 'nachname', 'alter'], [1, 'Bart', 'Simpson', 10],
 [2, 'Lisa', 'Simpson', 8], [3, 'Maggie', 'Simpson', 1]], 'WHERE',
('Bedingung', ['vorname', '=', 'Bart'])])
Keine weiteren Parses.
```

Man sieht, dass der semantische Wert des Spalten-Nonterminals korrekt in eine Liste der gewünschten Spalten übersetzt wird; ausserdem ist der semantische Wert des Tabellen-Nonterminals tatsächlich die gewünschte Tabelle. Als nächstes kümmern wir uns nun darum, die Tabelle auf die gewünschten Spalten zu reduzieren. Das erledigen wir in der Parse-Aktion zum SQL-Nonterminal:

```
def sql_action(select, spalten, fr, tabellen, where, bedingung):
   names = tabellen[0]
   namesidx = []
   # Finde die Spalten-Indices der zu behaltenden Spalten
   for spalte in spalten:
      idx = names.index(spalte)
      namesidx.append(idx)

   # Erstelle eine neue Tabelle, die nur die gewollten Spalten
   # enthält
   tabkopie = [ spalten ]
   for zeile in tabellen[1:]:
      zeilenkopie = []
      for idx in namesidx:
         zeilenkopie.append(zeile[idx])
      tabkopie.append(zeilenkopie)

   return tabkopie
```

Diese Parse-Aktion ist nun deutlich komplizierter als die vorangehenden. Sie kopiert die übergebene Tabelle, reduziert sie aber auf die angegebenen Spalten. Das Ergebnis lässt sich sehen (*demo/sql4.py*):

```
Abfrage: SELECT vorname , alter FROM personen WHERE vorname = 'Bart'
Aha, ich verstehe:  [['vorname', 'alter'], ['Bart', 10],
['Lisa', 8], ['Maggie', 1]]
Keine weiteren Parses.
```

Wir sind fast schon am Ziel: Der Parser liefert uns bereits eine Tabelle mit den gewünschten Spalten zurück. Allerdings liefert er immer noch alle Zeilen statt nur diejenigen, auf welche die Bedingung zutrifft. Das kommt daher, dass wir die Bedingung bisher völlig ignoriert haben. Das werden wir nun noch erledigen. Als erstes ändern wir die Grammatik ein wenig, um für die verschiedenen Bedingungs-Alternativen eigene Parse-Aktionen definieren zu können:

```
b1 = Nonterminal("b1")
b2 = Nonterminal("b2")
b3 = Nonterminal("b3")
bedingung >> b1 | b2 | b3
b1 >> T('name') + T('=') + T('string')
b2 >> T('name') + T('=') + T('number')
b3 >> T('name') + T('=') + T('name')
```

Nun sorgen wir dafür, dass der semantische Wert einer Bedingung immer aus einem Paar besteht, und zwar aus dem Namen der Spalte sowie einem Paar aus Typ (`string`, `number` oder `name`) und semantischem Wert des Teils auf der rechten Seite des Vergleichsoperators. Das sieht auf den ersten Blick unnötig kompliziert aus; wir benötigen aber den Typ des rechten Teils des Bedingungsausdrucks, um die Bedingung später korrekt auswerten zu können.

```
def b1_action(name, op, wert):
   return (name, ('string', wert))

def b2_action(name, op, wert):
   return (name, ('number', wert))

def b3_action(name, op, wert):
   return (name, ('name', wert))

def bedingung_action(b):
   return b
```

Als nächstes definieren wir eine Hilfsfunktion, die eine Tabellenzeile (also eine einzelne Liste) und eine Bedingung nimmt und `true` zurückliefert, wenn die Bedingung für diese Zeile erfüllt ist. Der Parameter `index` ist ein Python-Dictionary, das einen Spaltennamen in einen Listenindex übersetzt:

```
def check(zeile, index, name, wert):
   """Überprüft, ob der Wert in Spalte name = wert ist"""
   typ, wert = wert[0], wert[1]
   if typ == 'name':
      return zeile[index[name]] == zeile[index[wert]]
   else:
      return zeile[index[name]] == wert
```

Hier wird deutlich, weshalb wir den Typ des rechten Teils der Bedingung behalten haben – wir benötigen diesen Typ, weil der Typ `name` speziell behandelt werden muss: Damit lassen sich zwei Spalten in einer Tabelle auf Gleichheit prüfen. In den beiden anderen Fällen (`string` und `number`) brauchen wir jeweils nur den Wert in einer Tabellenspalte mit einem konstanten Wert zu vergleichen.

Diese `check`-Funktion können wir nun in die Parse-Aktion sql_action einbauen, so dass sie nur noch diejenigen Zeilen kopiert, auf welche die Bedingung zutrifft:

```
def sql_action(select, spalten, fr, tabellen, where, bedingung):
   tab = tabellen

   names = tab[0]
   index = makeColumnDict(names)
   namesidx = []
   # Finde die Spalten-Indices der zu behaltenden Spalten
   for spalte in spalten:
      namesidx.append(index[spalte])

   # Erstelle eine neue Tabelle, die nur die gewollten Spalten
   # enthält
   tabkopie = [ spalten ]
   for zeile in tab[1:]:
      if check(zeile, index, bedingung[0], bedingung[1]):
         zeilenkopie = []
         for idx in namesidx:
            zeilenkopie.append(zeile[idx])
         tabkopie.append(zeilenkopie)

   return tabkopie
```

Eine Bemerkung: das benötigte `index`-Dictionary wird mit der Hilfsfunktion `makeColumnDict` erzeugt.

Hier ist das Ergebnis unserer Bemühungen, nachdem wir sämtliche neuen Parse-Aktionen beim Parser registriert und die Hilfsfunktionen hinzugefügt haben (*demo/sql5.py*):

```
Abfrage: SELECT vorname , alter FROM personen WHERE vorname = 'Bart'
Aha, ich verstehe:  [['vorname', 'alter'], ['Bart', 10]]
Keine weiteren Parses.
```

Die SQL-Abfrage wird wie gewünscht in die Antwort `[['vorname', 'alter'], ['Bart', 10]]` übersetzt. Wir haben also in ungefähr 100 Zeilen Python eine (zugegeben sehr simple) SQL-Abfragesprache für Daten, welche wir in Python-Listen aufbewahren, geschaffen.

Um diese einfache Abfragesprache praktisch einsetzbar zu machen, müssten wir noch drei Erweiterungen implementieren:

Erstens müsste man in der FROM-Klausel mehr als eine Tabelle angeben können. Der semantische Wert dieser Klausel wäre dann das Vektorprodukt aller Tabellen: D.h. wir erstellen aus den einzelnen Tabellen eine grosse Tabelle, in der jede Zeile einer Tabelle mit allen anderen Zeilen der anderen Tabellen kombiniert wird. Das ist zwar ungeheuer ineffizient, aber einfach zu implementieren.

Zweitens müssten wir die WHERE-Klausel so erweitern, dass zusammengesetzte Bedingungen (mit logischem UND und ODER) möglich werden. Ansonsten können wir aus Tabellen-Joins nie mehr die gültigen Zeilen bestimmen.

Drittens müssten wir in der WHERE-Klausel nicht nur =, sondern weitere Verlgeichsoperatoren wie < oder > zulassen.

Keine dieser Erweiterungen ist besonders schwierig; sie sind dem Leser als Übung überlassen.

Schliesslich sei noch darauf hingewiesen, dass die vorliegende Implementation des eines SQL-ähnlichen Interpreters natürlich alles andere als effizient ist; es ging aber hier auch nur darum, den Einsatz eines Parsers an einem nichttrivialen Beispiel zu demonstrieren.