Tokenizer

Übersicht

- Was sind Tokenizer?
- Der Tokenizer von Covington
  - Aufrufdiagramm
  - Definition und Arbeitsweise der einzelnen Prädikate
  - Programmiertechnik *Look-Ahead*
- Wort- und Satzgrenzen erkennen
- Tokenisieren von Dateien
  - Probleme mit Dateiende und get_code/1 (bzw. get0/1)
- Auffangen von Exceptions mit catch/3
  - Programmiertechnik *Exception-Handling*

Motivation

Bisher: Linguistische Datenverarbeitung mit Prolog-Termen

```
?- phrase(s, [the,cat,sleeps]).
yes
```

Einlesen von Text

- einzelne ASCII-Zeichen
- Prolog-Terme einlesen

```
?- get_code(X).
|: a
X = 97 ?
yes
```

```
?- read(X).
|: [a,cat,sleeps].
X = [a,cat,sleeps]
yes
```

Mangel

Prolog hat keine vorgefertigte Eingabe-Möglichkeit, wo einfach ein Satz wie "The cat sleeps" zur Verarbeitung eingetippt bzw. eingelesen werden kann.

Zweck und Funktion eines Tokenizers

Wie sollen Benutzende einen Satz eingeben?

```
|: These are words.
```

Praktisch für Benutzende

```
[these, are, words,'.']
```

Praktisch zum Programmieren

Ein Tokenizer

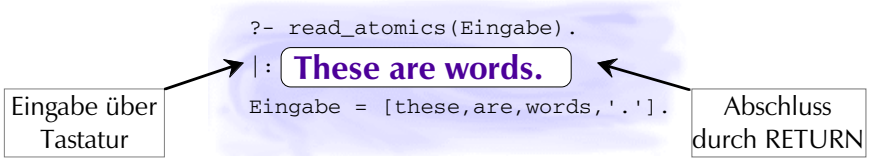
- **konsumiert** als Eingabe eine Sequenz von Zeichen (*Eingabestrom*)
- **gruppiert** die Eingabezeichen zu sinnvollen Einheiten (*Token*)
  - gewisse Eingabezeichen können dabei auch modifiziert werden
- **produziert** als Ausgabe die Sequenz der *Token (Liste)*

Ein einfacher Tokenizer

Ein Tokenizer findet sich in Covington (1994: Anhang B)

Aufruf des Tokenizers durch read_atomics/1

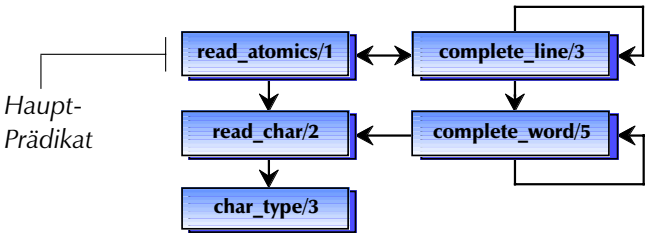
- liest eine Zeile voll Buchstaben von der Standardeingabe ein
- gibt Liste der Atome der tokenisierten Zeile als Resultat zurück
- Grossbuchstaben werden in Kleinbuchstaben verwandelt



Aufrufdiagramm des Tokenizers

Aufrufdiagramme (*call graphs*)

- Ein Prädikat ruft die Prädikate auf, zu denen ein Pfeil führt
  - *Eingebaute oder Standard-Prädikate werden meist weggelassen*
- Rekursive Prädikate sind durch Schlaufen verbunden



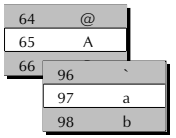
Aufrufdiagramm der Tokenizer Prädikate

Zeichen klassifizieren mit char_type/3

- **Das Hilfsprädikat char_type/3 klassifiziert Zeichen**
  - `end` — Zeilenende
  - `blank` — Leerzeichen
  - `alpha` — alphanumerische Zeichen, d.h. Buchstaben und Ziffern
  - `special` — übrige Zeichen

Zudem liefert char_type/3 das Zeichen als Kleinbuchstaben zurück.

```
?- char_type(65, Type, Char).
Type = alpha,
Char = 97
```



ASCII-Codes

char_type/3

```
char_type(10, end, 10) :- !.        % UNIX end of line mark
char_type(13, end, 13) :- !.        % Macintosh/DOS end of line mark
char_type(-1, end, -1) :- !.        % get0 end of file code

char_type(Code, blank, 32) :-       % blanks, other control codes
   Code =< 32, !.

char_type(Code, alpha, Code) :-     % digits
   48 =< Code, Code =< 57, !.

char_type(Code, alpha, Code) :-     % lower-case letters
   97 =< Code, Code =< 122, !.

char_type(Code, alpha, NewCode) :- % upper-case letters
   65 =< Code, Code =< 90, !,
   NewCode is Code + 32.            % translate to lower case

char_type(Code, special, Code).     % anything else
```


read_atomics/1 und read_char/2

- **read_atomics/1: Erstes Zeichen (*look ahead*) einlesen und dann den Rest verarbeiten lassen**
  - Die Vorausschau von einem Zeichen brauchts zum Entscheid, ob das aktuelle Zeichen das letzte eines Wort-Tokens ist.

```
read_atomics(Atomics) :-
  read_char(FirstChar, FirstType),
  complete_line(FirstChar, FirstType, Atomics).
```

- **read_char/2: Ein Zeichen einlesen und klassifizieren**

```
read_char(Char, Type) :-
  get_code(EnteredChar),
  char_type(EnteredChar, Type, Char).
```

complete_line/3

- **complete_line/3 besitzt folgende Argumente**
  - das *look-ahead*-Zeichen: *Integer*
  - dessen Typ (als Fallunterscheidung): *Atom*
    - falls `end`: stoppen, da Zeile fertig eingelesen ist!
    - falls `blank`: überspringen!
    - falls `alpha`: Zeichen zum Wort-Token kompletieren!
    - falls `special`: Zeichen zu eigenem Token machen!
  - Ergebnisliste aus den einzelnen Tokens: *Liste atomarer Terme*

```
?- complete_line(97, alpha, Result).
|: bba, baba.
Result = [abba, ',', baba, '.'].
```

Beispiel-Anfrage

96	`
97	a
98	b
99	c

ASCII-Codes

complete_line/3

```
complete_line(_, end, []) :- !.

complete_line(_, blank, Atomics) :-
  !,
  read_atomics(Atomics).

complete_line(Char, special, [A|Atomics]) :-
  !,
  name(A, [Char]),
  read_atomics(Atomics).

complete_line(FirstChar, alpha, [A|Atomics]) :-
  complete_word(FirstChar, alpha, Word, NextChar, NextType),
  name(A, Word),
  complete_line(NextChar, NextType, Atomics).
```

Rote oder grüne Cuts?

Unter der Voraussetzung, dass das 2. Argument beim Aufruf von complete_line/3 immer instanziiert ist: grün!

Spezifikation von complete_word/5

complete_word/5 besitzt folgende Argumente

- das *look-ahead*-Zeichen
- dessen Typ
- eine Liste, bestehend aus den ASCII-Codes der Zeichen, die zum gegenwärtigen Wort gehören
- dem nächstfolgenden Zeichen, das nicht zum Wort gehört,
- dessen Typ

96	`
97	a
98	b
99	c
58	:
59	;
60	<
	d
	e

ASCII-Codes

```
?- complete_word(97, alpha, List, FollowChar, FollowType).
|: bba;
List = [97, 98, 98, 97],
FollowChar = 59,
FollowType = special
```


Implementation von complete_word/5

Rekursionsschritt

- Look-ahead ist alphanumerisch!
  - FollowChar ist das zukünftige Look-ahead-Zeichen, das 1. Zeichen nach dem Wort!

```
complete_word(FirstChar, alpha, [FirstChar|List],
              FollowChar, FollowType) :-
  !, % red Cut
  read_char(NextChar, NextType),
  complete_word(NextChar, NextType, List, FollowChar, FollowType).
```

Abbruchbedingung

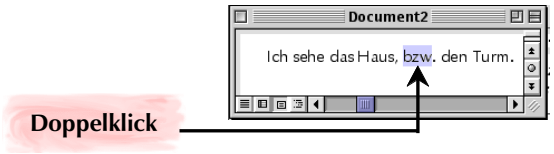
- Look-ahead ist *nicht* alphanumerisch!
  - Es wird nichts mehr konsumiert, nur noch Wort-Zeichen-Liste abgeschlossen!
  - Zukünftiger Look-Ahead wird auf aktuellen gesetzt!

```
complete_word(FirstChar, FirstType, [], FirstChar, FirstType).
```

Schwierigere Fälle

Welche Zeichen gehören zu welchem Token?

- 234.50
- Die Grille zirpt.
- Die Grille zirpt immer um 10.
- Die Grille zirpt immer am 10. Okt.
- Scarlett O'Hara sagte 'Schau mir in die Augen, Kleines' und erhielt dafür ca. Fr. 1'234.--.
- I said 'don't'
- …



Satzgrenzen erkennen

Bezeichnet ein Punkt das Ende eines Satzes?

- It was due Friday by 5 p.m. Saturday would be too late.
- She has an appointment at 5 p.m. Saturday to get her car fixed.

Lösungsansätze

- »Jeder Punkt ist ein Satzende!« — 8-45% Fehlerquote (Englisch)
- Abkürzungswörterbuch, Regeln mit regulären Ausdrücken — < 2%
- Training anhand Korpus — < 2%
- Lösungsansatz mit Neuronalem Netz Palmer/Hearst (1994) (*mit zusammenfassendem Einstieg ins Problem*)
- …

Zeilenweises Tokenisieren von Dateien

Erste Idee

- Datei besteht aus Folge von Zeilen
- Einlesen aller Zeilen durch *all-solution*-Prädikat

```
naive_tokenize_file(File, Lines) :-
    see(File),
    findall(Line, (repeat,read_atomics(Line)), Lines),
    seen.
```

- Wegen Determinismus von read_atomics/1 muss repeat/0 verwendet werden!

Aber

```
?- naive_tokenize_file('gedicht.txt', Lines).
! Existence error in get0/1
! attempt to read past end of stream
! goal:  get0('$stream'(1225984),_76)
```


Ausnahmefall: Lesen über Dateiende

Problem

- Dateiende darf nur einmal gelesen werden mit get_code/1!
- Bei einem weiteren Versuch wird eine *exception* ausgelöst!

```
?- tell('leer.txt'),told.
yes
?- see('leer.txt').
yes
?- get_code(C).
C = -1
yes
?- get_code(C).
! Existence error in get0/2
! attempt to read past end of stream
! goal:  get0('$stream'(1225984),_76)
```

Ausnahmefälle behandeln

Das Metaprädikat catch(*Goal,Pattern,Handler*) kann Ausnahmefälle auffangen.

- catch/3 ruft *Goal* auf
  - Falls *Goal* gelingt oder scheitert, macht catch/3 dasselbe.
  - Falls beim Beweis von *Goal* eine Exception *E* ausgelöst wird, passiert folgendes:
    - Falls *E* mit *Pattern* unifiziert werden kann, wird als neues Ziel *Handler* aufgerufen.
    - Falls *E* nicht mit *Pattern* unifiziert werden kann, wird *E* weiter hochgegeben.
- *exceptions* können beliebige Terme sein!
  - *exception* für das Lesen über das Dateiende (leicht systemabhängig):

```
existence_error(_,_,_,_,past_end_of_stream)
```


Zeilenweise Tokenisieren von Dateien

Idee

- Tokenizer soll scheitern, falls über Dateiende gelesen wird
- safe_read_atomics/1 liest via Backtracking alle Zeilen

```
safe_read_atomics(Atomics) :-
  catch(
    (repeat, read_atomics(Atomics)),              % Goal
    existence_error(_,_,_,_,past_end_of_stream), % Pattern
    fail                                          % Handler
  ).
```

- tokenize_file/2 liest alle Zeilen ein

```
tokenize_file(File, Lines) :-
   see(File),
   findall(Line, safe_read_atomics(Line), Lines),
   seen.
```


Literaturhinweise I

Tokenizer

- Michael A. Covington (1994): Natural Language Processing for Prolog Programmers. Prentice Hall.
  - Eine verbesserte Version mit etwas anderem Output findet sich unter http://www.ai.uga.edu/~mc/et/et.zip , bzw. Doku unter et.pdf
- Palmer, David D. (2000): Tokenisation and Sentence Segmentation. In: Handbook of natural language processing, edited by R. Dale, H. Moisl and H. Somers. New York. S. 11-35

Satzgrenzenerkennung

- David D. Palmer/Marti A. Hearst (1994): Adaptive Sentence Boundary Disambiguation. In: Proceedings of the ANLP '94, Stuttgart. http://xxx.lanl.gov/abs/cmp-lg/9411022

Literaturhinweise II

Programmieren mit Exceptions

- SICStus Prolog Handbuch: Dokumentation zu den folgenden Prädikaten für das Auffangen und Auslösen von Ausnahmen:
  - catch/3
  - throw/1

Tokenisieren mit andern Programmiersprachen

- Es kann sinnvoll sein, Prolog nur für strukturell komplexere Teile eines Programms einzusetzen und die Tokenisierung in einer Sprache zu realisieren, welche einfache und mächtige Stringbehandlung enthält.
  - siehe Programmiersprachen Perl, Ruby