Tokenizer

Übersicht

- Was macht ein Tokenizer?
- Zeichen klassifizieren
  - Trivial-Version für unseren Prolog-Tokenizer
  - Zeichen-Klassifikation in Unicode
- Arbeitsweise des vorgestellten Tokenizers
- Schwierigere Fälle; Satzgrenzen erkennen

Ziel

- Arbeitsweise des vorgestellten Tokenizers verstehen
  - *nicht:* Auswendiglernen dieses Prolog-Programms
- Wissen, dass das Problem nicht ganz trivial ist

Tokenizer

Ablauf diverser Anwendungen in diesem Semester:

- Benutzer gibt einen Satz ein
- Computer analysiert den Satz
- Computer führt irgendwelche Aktionen durch
  - Beantworten einer Frage durch Nachschlagen in Datenbank
  - Ausgabe der Phrasenstruktur
  - …

Tokenizer

Wie soll der Benutzer den Satz eingeben?

Dies ist ein Satz.

Praktisch für Benutzer

```
[dies, ist, ein,
 satz, '.']
```

Praktisch zum Programmieren

Ein Tokenizer

- nimmt eine Eingabe entgegen
- zerlegt die Eingabe in die einzelnen Wörter, Zahlen, Zeichen (»Tokens«)
- liefert die zerlegte Eingabe zurück, ggf. in normalisierter Form

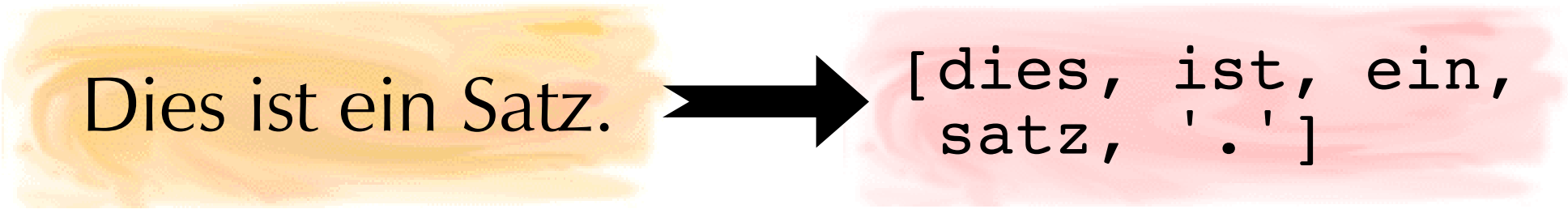
Tokenizer

Hanging Rock State Park
Victoria, Australien

Tokenizer

Wie bei den meisten Programmiersprachen ist auch bei Prolog kein Tokenizer eingebaut.

- Java: Tokenizer für Unicode-Texte
- MacOS: Finite-State-Tokenizer für div. Schriftsysteme



Wir müssen uns also einen eigenen entwickeln — oder einen vorhandenen übernehmen.

M. Covingtons Tokenizer

Ein Tokenizer findet sich in [Covington, 1994], Anhang B

- modifizierte Version auf den ausgeteilten Blättern abgedruckt
- http://www.coli.uni-sb.de/~brawer/prolog/tokenizer

Benutzung:

Eingabe über Tastatur

```
?- read_atomics(Eingabe).
|: Dies ist ein Satz.

Eingabe = [dies, ist, ein,
           satz, '.'].
```

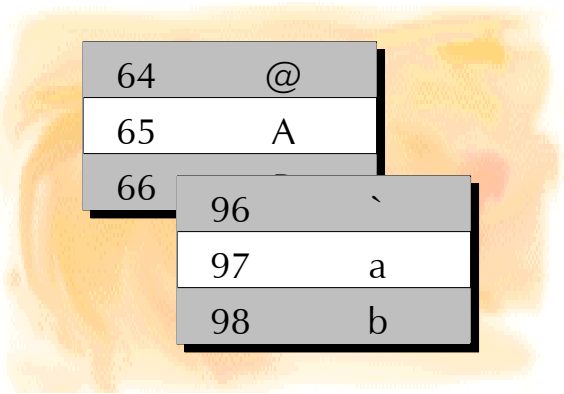
Tokenizer: Zeichen klassifizieren

Das Hilfsprädikat char_type/3 klassifiziert Zeichen:

- end — Zeilenende
- blank — Leerzeichen
- alpha — alphanumerische Zeichen, d.h. Buchstaben und Ziffern
- special — übrige Zeichen

Ausserdem liefert char_type das Zeichen als Kleinbuchstaben zurück.

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



ASCII-Codes

Zeichen-Klassifikation in Unicode

Der Unicode-Standard klassifiziert die Zeichen weitaus detaillierter:

- Mark
  - Mn *Mark, Non-Spacing* 355 U+0301 COMBINING ACUTE ACCENT
  - Mc *Mark, Sp. Combining* 85 U+094C DEVANAGARI VOWEL SIGN AU
  - Me *Mark, Enclosing* 6 U+20DD COMBINING ENCLOSING CIRCLE

Code	*Kategorie*	*Anzahl*	*Code Bsp.-Z.*	*Bezeichnung Beispiel-Zeichen*

- Number
  - Nd *Number, Decimal Digit* 159 U+0034 DIGIT FOUR
  - Nl *Number, Letter* 45 U+216B ROMAN NUMERAL TWELVE
  - No *Number, Other* 170 U+00BC VULGAR FRACTION ONE QUARTER

Zeichen-Klassifikation in Unicode

- Separator
  - Zs *Separator, Space* 16 U+00A0 NO-BREAK SPACE
  - Zl *Separator, Line* 1 U+2028 LINE SEPARATOR
  - Zp *Separator, Paragraph* 1 U+2029 PARAGRAPH SEPARATOR
- Letter
  - Lu *Letter, Uppercase* 693 U+0041 LATIN CAPITAL LETTER A
  - Ll *Letter, Lowercase* 809 U+0077 LATIN SMALL LETTER W
  - Lt *Letter, Titlecase* 4 U+01C8 LATIN CAPITAL LETTER L WITH SMALL LETTER J
  - Lm *Letter, Modifier* 46 U+02C0 MODIFIER LETTER GLOTTAL STOP
  - Lo *Letter, Other* 34569 U+05D0 HEBREW LETTER ALEF

Zeichen-Klassifikation in Unicode

- Punctuation
  - Pc *Punctuation, Connector* 9 U+005F LOW LINE
  - Pd *Punctuation, Dash* 15 U+002D HYPHEN MINUS
  - Ps *Punctuation, Open* 44 U+0028 LEFT PARENTHESIS
  - Pe *Punctuation, Close* 39 U+0029 RIGHT PARENTHESIS
  - Po *Punctuation, Other* 133 U+0021 EXCLAMATION MARK
- Symbol
  - Sm *Symbol, Math* 289 U+002B PLUS SIGN
  - Sc *Symbol, Currency* 26 U+0028 DOLLAR SIGN
  - Sk *Symbol, Modifier* 60 U+005E CIRCUMFLEX ACCENT
  - So *Symbol, Other* 1296 U+00A9 COPYRIGHT SIGN

Zeichen-Klassifikation in Unicode

- Other
  - Cc *Other, Control* 65 U+001B — (Escape)
  - Cf *Other, Format* 16 U+200C ZERO WIDTH NON-JOINER
  - Cs *Other, Surrogate* 2048 U+DC00 —
  - Co *Other, Private Use* 6400 U+E000 —
  - Cn *Other, Not Assigned* 0 — —

Diese Zahlen stimmen für Version 2.0.14.

Aktuelle Daten jeweils auf http://www.unicode.org

Tokenizer: Arbeitsweise

Ein einzelnes Zeichen wird mit get0/1 eingelesen.

```
read_char(Char, Type) :-
  get0(EnteredChar),
  char_type(EnteredChar, Type, Char).
```

Covingtons Tokenizer arbeitet mit einem sogenannten *Look-Ahead:*

- Das Programm schaut immer ein Zeichen »in die Zukunft«.

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

Tokenizer: Arbeitsweise

complete_line/3 besitzt folgende Argumente:

- das gegenwärtige Zeichen
- dessen Typ
- eine Liste mit dem Ergebnis, d.h. den einzelnen Tokens

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

Beispiel-Anfrage

96	`
97	a
98	b
99	c

ASCII-Codes

Tokenizer: Arbeitsweise

```
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).
```

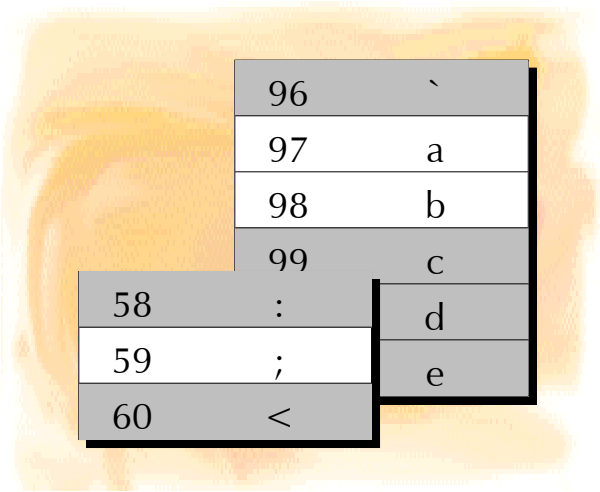
Cut

»Dies ist der richtige Pfad; andere Lösungen brauchen nicht gesucht zu werden.«

Tokenizer: Arbeitsweise

complete_word/5 besitzt folgende Argumente:

- das gegenwärtige Zeichen
- dessen Typ
- eine Liste, bestehend aus den ASCII-Codes der Zeichen, die zum gegenwärtigen Wort gehören
- dem nächstfolgenden Zeichen
- dessen Typ



ASCII-Codes

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

Tokenizer: Arbeitsweise

Rekursiver Fall

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

Abbruchbed.

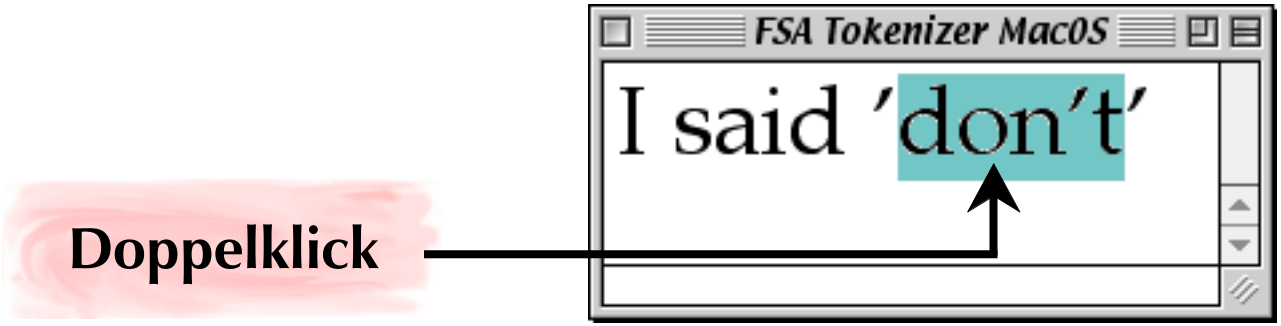
≠ alpha

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

Schwierigere Fälle

Welche Zeichen gehören zum Token?

- 234.50
- Die Grille zirpt.
- Die Grille zirpt immer um 10.
- Die Grille zirpt immer am 10. Oktober.
- Scarlett O'Hara sagte 'Schau mir in die Augen, Kleines' und erhielt dafür ca. Fr. 1'234.50.
- 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« — 10% Fehlerquote (Englisch)
- Abkürzungswörterbuch, Regeln (riesiger Aufwand) — ~2%
- Training anhand Korpus — ~2%
- Lösungsansatz mit Neuronalem Netz (guter Einstieg in Problem)
  - David D. Palmer/Marti A. Hearst: Adaptive Sentence Boundary Disambiguation. In *Proceedings of the ANLP '94*, Stuttgart, Oktober 1994. http://xxx.lanl.gov/abs/cmp-lg/9411022

tokenizer.pl

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999
Online unter http://www.coli.uni-sb.de/~brawer/prolog/tokenizer/

```
/*********************************************************
 * read_atomics(-Atomics)
 *
 *  Reads a line of text, breaking it into a list of atomic
 *  terms.
 *
 *  Example: "This is an example." [this,is,an,example,'.'].
 *
 *  Source: [Covington, 1994], Appendix B
 *********************************************************/

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



% read_char(-Char, -Type)
%  Reads a character and runs it through char_type/1.

read_char(Char, Type) :-
   get0(EnteredChar),
   char_type(EnteredChar, Type, Char).



% complete_line(+FirstChar, +FirstType, -Charlists)
%  Given FirstChar (the first character) and FirstType (its type),
%  reads and tokenizes the rest of the line into atoms and numbers.

complete_line(_, end, []) :- !.                  % stop at end

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

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

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

```
% complete_word(+FirstChar, +FirstType,
%               -List, -FollowChar, -FollowType)
%  Given FirstChar (the first character) and FirstType (its type),
%  reads the rest of a word, putting its characters into List.

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

complete_word(FirstChar, FirstType, [], FirstChar, FirstType).
   % where FirstType is not alpha; otherwise, the first clause
   % for complete_word would have been taken.


% char_type(+Code, ?Type, -NewCode)
%  Given an ASCII code, classifies the character as
%  'end' (of line/file), 'blank', 'alpha'(numeric), or 'special'.
%  and changes it to a potentially different character (NewCode).

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).
```

Aufgaben: Tokenizer

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999

1. Umgang mit Prolog

Während die Übungen des letzten Semesters eventuell noch auf dem Papier zu lösen waren, wirst Du dieses Semester kaum ohne Prolog-Interpreter auskommen können. Stelle sicher, dass Du ein Prolog zur Verfügung hast und damit umgehen kannst.

2. Repetition

Mache Dir als Repetition anhand der Unterlagen des letzen Semesters klar, wie Prolog-Prädikate abgearbeitet werden.

3. Tokenizer

Auf einem separaten Blatt ist der Tokenizer aus [Covington, 1994], Anhang B, wiedergegeben.

a) Lade das Programm in Deinem Computer, entweder von der Internet-Adresse http://www.coli.uni-sb.de/~brawer/prolog/ – oder sonst durch Abtippen.

b) Probiere den Tokenizer aus, indem Du read_atomics/1 einige Male aufrufst und einige Sätze eingibst.

c) Mache Dir klar, wie dieser Tokenizer funktioniert.

d) Wie wird die Eingabe »Blöd blöken die Frühlingsblüten.« in Tokens zerlegt? Welche Änderungen wären nötig, um die deutschen Umlaute korrekt zu verarbeiten? Du darst selbstverständlich auch einen intelligenteren Satz eingeben.

4. Tokenizer von Textverarbeitungen

Starte eine Textverarbeitung Deiner Wahl und versuche mit Doppelklicken herauszufinden, wie ihr Tokenizer arbeitet. Finde Beispiele, wo der Tokenizer die Wortgrenzen falsch erkennt.

Wenn Deine Textverabeitung eine Grammatikkorrektur besitzt, gib einen Satz mit einem Fehler ein, welcher dazu führt, dass der gesamte Satz markiert wird (Microsoft Word macht dies beispielsweise bei sehr langen, komplexen Sätzen). Plaziere nun an geeigneter Stelle im Satz Abkürzungen usw., um herauszufinden, ob (und wann) die Textverarbeitung irrtümlicherweise Satzgrenzen setzt.