Tokenization using DCG Rules

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602-7415 U.S.A.

2000 April 21

1 Overview

This paper presents a tokenizer for English text, including numerals, that is implemented in Prolog using DCG rules. It is thus a top-down recursive-descent parser rather than a finite-state automaton. The lower efficiency of this parsing algorithm does not seriously degrade performance, and it makes the tokenizer much easier to modify for unusual kinds of input tokens.[1]

2 Introduction

Tokenization is the process of converting a string of characters into a list of words and other significant elements. For example, the string

```
We owe $1,000,000 to Agent 007.
```

can be tokenized to yield this list of Prolog atoms and numbers:

[1]I want to thank Sharon Covington for helping me check this paper for typing errors. All responsibility for remaining errors is my own.

This is an example of a student project paper for CSCI 8570. Students, please note that Covington (1994) appears in the bibliography only because it is the only source of a vital piece of information (the character type predicate used in the program). You should not cite textbooks unless you are actually using information that is not available anywhere else.

```
[we,owe,'$',1000000,to,agent,7]
```

Note that in this example, `007` in the input is interpreted as the number 7, which may or may not be desirable behavior.

Tokenization is needed both for natural language processing and for the compilation of programming languages. The latter is a more mature technology with a larger literature.

The syntax of programming languages, including the structure of tokens, is almost always specified by using context-free phrase-structure rules (see for example Deransart et al. 1991, Jensen and Wirth 1974, or practically any official language specification). However, the actual work of tokenization is normally done with finite-state transition networks equivalent to regular expressions (Aho and Ullman 1977, Beesley and Karttunen forthcoming, Mohri 1996).

Until recently, it was taken for granted that phrase-structure rules were not, in themselves, executable. However, Prolog makes them executable in the form of definite clause grammar (DCG) rules (O'Keefe 1990, Shalfield 1999).[2] For example, the rule

$A \rightarrow B\ C$

can go into Prolog as

```
a --> b, c.
```

which is translated by the Prolog compiler into

```
a(L1,L) :- b(L1,L2), c(L2,L).
```

or the equivalent.

Thus the stage is set for building tokenizers by writing phrase-structure rules. This project presents a tokenizer of that type for natural language.

3 The grammar of tokens

Except for the more complex kinds of numerals, the grammar of tokens is simple. A token consists of a continuous series of letters, a continuous series of digits, or a single special character (punctuation mark or the like).

[2]Despite almost universal availability, DCG rules are not part of the ISO Prolog standard (Deransart et al. 1996).

Whitespace characters (spaces, tabs, and line breaks) that occur between tokens are skipped. These facts are summarized by these grammar rules:

$$tokenlist \rightarrow whitespace\ (token\ tokenlist)$$

$$whitespace \rightarrow \left\{ \begin{array}{c} whitespacecharacter\ whitespace \\ \emptyset \end{array} \right\}$$

$$token \rightarrow word$$

$$token \rightarrow numeral$$

$$token \rightarrow special$$

$$word \rightarrow letter\ (word)$$

$$numeral \rightarrow digit\ (numeral)$$

$$special \rightarrow specialchar$$

Note that *whitespace* can be of zero length, and therefore so can *tokenlist*. (An empty tokenlist results from tokenizing an empty string.) However, *word* and *numeral* must each have at least one character, and *special* is exactly one character.

4 Translating the rules into DCG

Crucially, the DCG rules must not only find the tokens, but also deliver them, in some useful form (e.g., as lists of characters) as the output of the process. Thus, the DCG version of these grammar rules must include arguments in which to collect the data.

In addition, because DCG makes no provision for optional elements, some of the phrase-structure rules above must be split into more than one DCG rule.

Finally, tokenization must be deterministic; that is, there is only one way to tokenize a string. Accordingly, if the grammar rules provide alternatives, the tokenizer must execute a cut to remove each backtrack point promptly after generating it.

With these considerations in mind, the middle block of grammar rules, defining *word*, *numeral*, and *special*, go into Prolog straightforwardly:

```
word([L|Rest]) --> letter(L),word(Rest).
word([L]) --> letter(L).

numeral([D|Rest]) --> digit(D),numeral(Rest).
numeral([D]) --> digit(D).

special([S]) --> specialchar(S).
```

Each rule has an argument, a list in which the characters of the token accumulate.

Note that some backtracking is unavoidable; on the last character of each word or numeral, the tokenizer has to backtrack after discovering that there are no more characters following.

The definition of *tokenlist* includes the cut. Together with the definition of *whitespace* (here called `blank0` meaning "zero or more blanks"), it is:

```
token_list([T|Rest]) --> blank0,token(T),!,token_list(Rest).
token_list([]) --> blank0.

blank0 --> [C],{char_type(C,blank)},!,blank0.
blank0 --> [].
```

Here `blank0` has no argument because whitespace characters need not be collected into a list.

The rules that define *token* convert the various kinds of tokens into Prolog atoms or numbers. They are:

```
token(T) --> special(L), {atom_codes(T,L)}.
token(T) --> word(W),    {atom_codes(T,W)}.
token(T) --> numeral(N), {number_codes(T,N)}.
```

5 Character types

It remains to define *specialchar*, *letter*, and *digit*. This could be done by writing rules such as:

```
letter("a") --> "a".
letter("b") --> "b".          % etc.
```

```
digit("0") --> "0".
digit("1") --> "1".            % etc.

specialchar(".") -->  ".".
specialchar("!") -->  "!".     % etc.
```

However, the actual program uses an adaptation of `char_type` from Covington (1994, p. 320), which classifies characters by doing arithmetic on their ASCII codes. For example, all codes between 97 and 122, inclusive, are lowercase letters. Capital letters are converted to lower case during classification.

6 More elaborate numerals

Numerals are not just strings of digits. The digits to the left of the point are normally (thought not invariably) divided by commas into groups of three, and there may be a decimal point followed by subsequent digits. Accordingly, the tokenizer uses the following grammar for numerals:

numeral $\rightarrow$ ',' *digit digit digit* (*numeral*)

numeral $\rightarrow$ *digit* (*numeral*)

numeral $\rightarrow$ '.' *digitstring*

digitstring $\rightarrow$ *digit* (*digitstring*)

Here *digitstring* is a string of digits without commas or decimal points — the old definition of *numeral* — and *numeral* is now defined more elaborately. In addition to accepting all normal notations, these rules also accept some nonstandard ways of writing numerals, such as "34242424,234.0" (with inconsistent use of commas). However, "24." (meaning 24.0, as in FORTRAN programming) is not accepted.

For the Prolog implementation of these rules, see the program listing.

7 Evaluation

In its present form, this tokenizer is arguably quite inefficient. It backtracks once on the last character of every word or numeral. It spends time creating

other backtrack points that have to be thrown away by the cut. Even O'Keefe (1990), an enthusiastic user of DCG rules, does not use them for tokenization (Chapter 10).

However, the ease of modifying the rules makes this tokenizer ideal for experimental work, and performance is quite acceptable in all cases tried so far.

References

Aho, Alfred V., and Ullman, Jeffrey D. (1977) *Principles of compiler design.* Reading, Mass.: Addison-Wesley.

Beesley, Kenneth R., and Karttunen, Lauri (forthcoming) *Finite-state morphology: tools and techniques.* Draft online at http://www.cis.upenn.edu/~cis639/docs/book.ps.

Covington, Michael A. (1994) *Natural language processing for Prolog programmers.* Englewood Cliffs, N.J.: Prentice Hall.

Deransart, P.; Ed-Dbali, A.; and Cervoni, L. (1996) *Prolog: the standard.* Berlin: Springer.

Jensen, Kathleen, and Wirth, Niklaus (1974) *Pascal user manual and report.* Second edition. Berlin: Springer.

Mohri, Mehryar (1996) On some applications of finite-state automata theory to natural language processing. Online at http://www.research.att.com/~mohri/jnle.ps.gz.

O'Keefe, Richard A. (1990) *The craft of Prolog.* Cambridge, Mass.: MIT Press.

Shalfield, Rebecca (1999) *LPA Win-Prolog 4.0 programming guide.* London: Logic Programming Associates.

Program Listing

```
% File sampleproj.pl - M. Covington - 2001 April 21
% A tokenizer using DCG rules.
```

```
:- use_module(library(lists)).    % provides append/3 in SICStus Prolog


% A test predicate to demonstrate that it works

test :-  token_list(What,
             " We owe $1,048,576.24 to Agent 007 for Version 3.14159! ",[]),
         write(What), nl,
 write('[There should not be any alternatives here...]'), nl,
         fail.


% A token list is a series of zero or more tokens.
% Its argument consists of the list of tokens, as atoms and numbers.
% The cut ensures that the maximum number of characters is
% gathered into each token.
%
% To tokenize a string, do this:  ?- token_list(Result," the string ",[]).
%

token_list([T|Rest]) --> blank0,token(T),!,token_list(Rest).
token_list([]) --> blank0.

% blank0 is a series of zero or more blanks.

blank0 --> [C],{char_type(C,blank)},!,blank0.
blank0 --> [].

% Several kinds of tokens.
% This is where lists of characters get converted into atoms or numbers.

token(T) --> special(L), {atom_codes(T,L)}.
token(T) --> word(W),    {atom_codes(T,W)}.
token(T) --> numeral(N), {number_codes(T,N)}.

% A word is a series of one or more letters.
% The rules are ordered so that we first try to gather as many
% characters into one digit_string as possible.

word([L|Rest]) --> letter(L),word(Rest).
word([L]) --> letter(L).

% A numeral is a list of characters that constitute a number.
% The argument of numeral(...) is the list of character codes.
```

```
numeral([C1,C2,C3|N]) --> ",", digit(C1), digit(C2), digit(C3), numeral(N).
numeral([C1,C2,C3])   --> ",", digit(C1), digit(C2), digit(C3).

numeral([C|N]) --> digit(C), numeral(N).    % multiple digits

numeral([C])   --> digit(C).                % single digit

numeral(N)     --> decimal_part(N).         % decimal point and more digits

decimal_part([46|Rest]) --> ".", digit_string(Rest).

digit_string([D|N]) --> digit(D),digit_string(N).
digit_string([D]) --> digit(D).

% Various kinds of characters...

digit(C)        --> [C], {char_type(C,numeric)}.

special([C])    --> [C], {char_type(C,special)}.

letter(C)       --> [C], {char_type(C,lowercase)}.

letter(C)       --> [U], {char_type(U,uppercase), C is U+32}.
                                    % Conversion to lowercase


% char_type(+Code,?Type)
%   Classifies a character (ASCII code) as
%   blank, numeric, uppercase, lowercase, or special.
%   Adapted from Covington 1994.

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

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

char_type(Code,Y):-% lowercase letters
97 =< Code,
Code=< 122,!,
Y = lowercase.
```

```
char_type(Code,Y):-% uppercase letters
65 =< Code, Code=< 90,!,
        Y = uppercase.

char_type(_,special).    % all others

% End of sampleproj.pl
```