ET: an Efficient Tokenizer in ISO Prolog

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602-7415 U.S.A.
http://www.ai.uga.edu/mc

2003 February 12

1 Introduction

This paper describes *ET,* an efficient tokenizer written in ISO standard Prolog and fully compatible with SWI-Prolog version 5.

Tokenization is the process of breaking a text file up into words and/or other significant units. For example, *ET* breaks the input string

```
Joe's brother doesn't owe us $1,234,567.89.
```

into the series of tokens:

```
[
  w([j,o,e]),
  w([s]),
  w([b,r,o,t,h,e,r]),
  w([d,o,e,s,n,t]),
  w([o,w,e]),
  w([u,s]),
  s('$'),
  n(['1','2','3','4','5','6','7','.','8','9']),
  s('.')
]
```

Each token is tagged as `w(...)` (word), `n(...)` (number), or `s(...)` (special character). Words and numbers are represented as lists of one-character atoms; special characters stand by themselves. The white space (blanks, line breaks etc.) between the tokens is skipped.

In words, all letters are translated to lower case. This causes little or no loss of information, since case distinctions are necessary in English only in a few rare contexts.[1]

[1]Such as *deaf* 'unable to hear' vs. *Deaf* 'belonging to a community of deaf people, using sign language, and so forth.'

2 Apostrophes

Apostrophes are treated as whitespace characters, except that in the context ϕ*'t*, where ϕ is any letter, the apostrophe is simply skipped, as if it where not there.

This seemingly odd behavior is motivated by the fact that, in English, apostrophes are placed at morpheme boundaries except in the context *n't* (for *not*) (Bunčić 2002a,b). Thus, we want to break up *they've*, *she'll*, and *boy's* while leaving *doesn't*, *won't*, etc. unbroken.

When the possessive suffix is unpronounced (as in *boys'*), it simply disappears. This is no catastrophe because it is already unobservable in the spoken language. Single quotation marks, indistinguishable from apostrophes, also disappear.

Tokenizer efficiency is also a factor Without increasing the lookahead or adding state variables, the tokenizer cannot recognize *n't*, but it can recognize ϕ*'t* by looking ahead to *'t* at a time when it knows it is processing a letter. Nor can it split off *n't* as a separate word because, at the time, it has already processed the *n*.

A few words, such as *ma'am* and *fo'c'sle,* end up with spurious breaks, but they can be rejoined at a later stage of processing.

3 Number tokens

Number tokens consist of digits, commas (which are omitted from the output), and decimal points (which are preserved). The tokenizer does not attempt to convert number tokens to actual numbers; that is a job for subsequent processing. Also, some number tokens are not really numbers at all (e.g., *2.3.6* in the phrase *section 2.3.6 of the book*).

Notice that the period (`.`) is part of a numeric token if it is followed by a digit, but a special character otherwise. Thus, in tokenizing

```
He doesn't owe us $1,234,567.89.
```

the tokenizer cannot decide how to handle the digit `9` until it has looked ahead at the next *two* characters, the period and the subsequent white space.

In addition, strings such as

```
.03
```

are treated as numeric.

Because two characters of lookahead are needed in this context, the tokenizer handles lookahead in two ways. It normally reads one character beyond the character currently being handled ("read-ahead"). The second character of lookahead, when needed, is achieved by using `peek_char/1`, a built-in predicate that can always see one character beyond the last character already read.

4 Design goals

ET is designed to be efficient. Specifically:

- It is deterministic (never backtracks any substantial distance);
- It uses no more lookahead than actually necessary;
- It constructs no unneeded data structures;
- It does all computations as quickly as possible;
- It never discards information that may be needed later (that's why it tags all the tokens with `w()`, `s()`, or `n()`);
- As far as possible, it never computes the same information more than once.

ET can tokenize 1 megabyte of typical text in about 7 seconds on a 1-GHz Pentium.[2]

5 User-callable predicates

The following predicates are normally called by the user of ET:

tokenize_file(+Filename,−Tokens)

Reads an entire text file and returns it as a stream of tokens. Sample query:

```
?- tokenize_file('c:\\temp\\myfile.txt',X).
X = [w([h,e]),w([d,o,e,s,n,t]),w([o,w,e]),w([u,s]),s('$'), ... ]
```

tokenize_stream(+Stream,−Tokens)

Like `tokenize_file`, but the first argument is an open stream rather than a filename. The user is responsible for opening and closing the stream. Sample query (equivalent to the one above):

```
?- open('c:\\temp\\myfile.txt',read,S),
   tokenize_stream(S,X),
   close(S).

X = [w([h,e]),w([d,o,e,s,n,t]),w([o,w,e]),w([u,s]),s('$'), ... ]
```

tokenize_line(+Stream,−Tokens)

Like `tokenize_stream`, but reads only one line of the file, and can be called again to read the next one. If there are no more tokens, an empty list is returned. It is permissible to call `tokenize_line` repeatedly at end of file, and an empty list will be returned each time.

[2]Note that, by default, SWI-Prolog does not allocate enough memory to hold a text of this size in tokenized form. The allocation can easily be increased. On the AI Center's "Lend-Out Disc" dated January 2003, the file `default.el` greatly increases the memory allocation for SWI-Prolog under Emacs.

Hint: In SWI-Prolog, the stream `user` is always open to the keyboard. Hence, you can tokenize a line from the keyboard by doing this:

```
?- tokenize_line(user,X).
This is a test.
X = [w([t,h,i,s]),w([i,s]),w([a]),w([t,e,s,t]),s('.')]
```

tokenize_line_dl(+Stream,−Tokens/Tail)

Like `tokenize_line`, but the output is a difference list. (Normally `Tail` is uninstantiated.) This allows you to concatenate the tokens from two or more lines of input without using `append`. Here is a simple example, reading from the keyboard:

```
?- tokenize_line_dl(user,X/Y).
This is a test.
X = [w([t,h,i,s]),w([i,s]),w([a]),w([t,e,s,t]),s('.')|Y]
```

(If you actually try this in SWI-Prolog, the output will be slightly more elaborate, with Y equated to an uninstantiated memory location denoted by something like `_G123`. The effect is the same.)

Here is a more elaborate example, reading two lines of input from the keyboard and concatenating them by difference-list instantiation:

```
?- tokenize_line_dl(user,X/Y), tokenize_line_dl(Y/Z).
Hello.
Look at this.
X = [w([h,e,l,l,o]),s('.'),w([l,o,o,k]),w([a,t]),w([t,h,i,s]),s('.')|Z]
Y = [w([l,o,o,k]),w([a,t]),w([t,h,i,s]),s('.')|Z]
```

(Again, in SWI-Prolog, the uninstantiated locations will have designations such as `_G001`, but the effect is the same.)

Note that if you want a proper list rather than a difference list, all you have to do is instantiate the last tail (here `Z`) to `[]`.

The main recursive loop in *ET* uses `tokenize_line_dl` to read the file one line at a time and concatenate the lists of tokens it obtains each time.

tokens_words(+Tokens,−Words)

Takes a list of tokens, and extracts just the words, converting them to Prolog atoms. Example:

```
?- tokens_words( [s('"'),w([t,h,i,s]),w([i,s]),w([e,x,a,m,p,l,e]),n([1])], What ).
What = [this,is,example]
```

This is something you're likely to want to do if you are simply counting words without doing any morphological analysis.

6 Character classification

ET is distributed as source code and designed to be modified by the user. The part you are most likely to modify is the character classification table, which classifies characters as null (ignored on input), "eol" (end of line), whitespace, alphabetic, numeric, or special, and also translates uppercase letters to lowercase (e.g., `A` to `a`). In the current version of *ET,* it consists of two predicates:

```
char_type_char(Char,Type,Tr) :-
   char_table(Char,Type,Tr),
   !.

char_type_char(Char,special,Char).

char_table(' ',     whitespace,  ' ').     % blank
char_table('\t',    whitespace,  ' ').     % tab
char_table('\r',    whitespace,  ' ').     % return

char_table(end_of_file, eol, end_of_file).
char_table('\n',        eol, '\n'       ). % new line mark

char_table(' ',     whitespace,  ' ').     % blank
char_table('\t',    whitespace,  ' ').     % tab
char_table('\r',    whitespace,  ' ').     % return

char_table(a,     letter,    a ).
char_table(b,     letter,    b ).
char_table(c,     letter,    c ).
...
char_table('A',   letter,    a ).
char_table('B',   letter,    b ).
char_table('C',   letter,    c ).
...
char_table('0',   digit,     '0' ).
char_table('1',   digit,     '1' ).
char_table('2',   digit,     '2' ).
char_table('3',   digit,     '3' ).
char_table('4',   digit,     '4' ).
char_table('5',   digit,     '5' ).
char_table('6',   digit,     '6' ).
char_table('7',   digit,     '7' ).
char_table('8',   digit,     '8' ).
char_table('9',   digit,     '9' ).
```

Here `char_type_char` takes a character and gives its type and its translation (i.e., conversion to lowercase). Thus:

```
?- char_type_char('A',Type,Char).
Type = letter
Char = a
```

All characters not listed in `char_table` are classified as special, The character table itself, `char_table`, uses indexing to look up each character as quickly as possible. Its first entry, `end_of_file`, is the atom returned by `get_char` and `peek_char` when they encounter the end of the file. All the other entries are one-character atoms. The program might be even faster if `char_type_char` were eliminated and a cut were put in each clause of `char_table`, followed by a catch-all clause at the end to handle atoms that are not in the table.

7 Program logic

As noted already, the higher-level predicates such as `tokenize_file` and `tokenize_stream` ultimately pass control to `tokenize_line_dl`, which tokenizes a line into a difference list. When tokenizing more than one line, the difference lists are automatically concatenated by instantiation:

```
tokenize_stream(Stream,Tokens) :-
   tokenize_line_dl(Stream,Tokens/Tail),
   tokenize_stream(Stream,Tail).
```

The recursion in this clause demonstrates an important technique and should be studied carefully.

The tokenizer itself is a finite-state transition network with two characters of lookahead. That is, it spends all its time making decisions and jumping from one state (and hence one clause) to another. All the recursion is tail recursion; that is, the algorithm is not inherently recursive and does not process structures within structures of the same type. After processing every token, the tokenizer moves on to the next without needing to remember how it got there.

The two characters of lookahead are achieved by reading one character ahead, then using `peek_char` to see a second character when necessary.

The process of tokenizing a line is launched as follows:

```
tokenize_line_dl(Stream,Dlist) :-
   get_char_and_type(Stream,Char,Type),
   tokenize_line_x(Type,Char,Stream,Dlist).
```

Here `get_char_and_type` gets a character, then classifies and translates it according to the character table, skipping apostrophes on input. The character and type are passed to `tokenize_line_x`, which will decide what to do from there.

In turn, `tokenize_line_x` has seven clauses, depending on whether `Char` is an end of line mark, a whitespace character, a letter, a digit, a period followed by a digit (which is special because it can be the beginning of a decimal number), a special character, or something else (which should not happen). An end of line mark terminates processing. A whitespace

character is skipped, and a special character is a token by itself; each of these can be dealt with immediately, and then the tokenizer will look for another token.

In the other situations — a letter, a digit, or a period introducing a number — the token may consist of more than one character, so `tokenize_line_x` has to hand off control to another predicate to complete the token. For example, here is how a word (a series of letters) is tokenized:

```
tokenize_line_dl(Stream,Dlist) :-
   get_char(Stream,C),
   char_type_char(C,Type,Char),
   tokenize_line_x(Type,Char,Stream,Dlist).

...

tokenize_line_x(letter,Char,Stream,[w(T)|Tokens]/Tail) :-
   !,
   tokenize_letters(letter,Char,Stream,T,NewType,NewChar),
   tokenize_line_x(NewType,NewChar,Stream,Tokens/Tail).

...

tokenize_letters(letter,Char,Stream,[Char|Rest],NewType,NewChar) :-
   % It's a letter, so process it, read another character ahead, and recurse.
   !,
   get_char(Stream,C),
   char_type_char(C,Type2,Char2),
   tokenize_letters(Type2,Char2,Stream,Rest,NewType,NewChar).

tokenize_letters(Type,Char,_,[],Type,Char).
   % It's not a letter, so don't process it; pass it to the calling procedure.
```

That is: `tokenize_line_dl` reads a letter and passes it to the clause of `tokenize_line_x` shown here. That clause, in turn, passes control to `tokenize_letters`, which adds the letter to the token, reads another character, and tries to continue doing the same thing. But if the subsequent character is not a letter, `tokenize_letters` stops recursing and passes control back to `tokenize_line_x`.

8 Limitations

The current version of *ET* makes no attempt to handle non-ASCII characters correctly. It could easily be extended to handle ANSI or even Unicode.[3] Nor does it behave intelligently

[3]Extending it to handle all of Unicode would be "easy" in the theoretical sense, but nonetheless a gigantic amount of work!

when encountering mixed letters and digits, such as part numbers, library call numbers (such as *QA76.93.B234*), file names, or URLs.

ET assumes that a subsequent step in the calling program, presumably the morphological analyzer, will determine whether numeric tokens actually represent numbers, and if so, convert them to numbers using `number_chars` or a similar predicate.

References

[1] Bunčić, Daniel (2002a) Apostrophe rules. *LINGUIST List,* May 31, 2002. http://www.linguistlist.org/issues/13/13-1566.html.

[2] Bunčić, Daniel (2002b) The apostrophe: a neglected and misunderstood reading aid. Presented at the Third International Workshop on Writing Systems, Köln. Available online at http://www.uni-bonn.de/∼dbuncic/apostroph/.