Prolog(Elex): A New Tool to Generate Prolog Tokenizers

Gertjan van Noord
Alfa-Informatica & BCN
University of Groningen
vannoord@let.rug.nl

Abstract

This paper presents a tool called *Elex(Prolog)* to construct tokenizers (lexical analysers, scanners, lexers) in Prolog. It is based on *Elex*, a multilingual scanner generator by Matthew Phillips. The paper motivates the tool, and presents its functionality and implementation. It also compares *Elex(Prolog)* to the only alternative Prolog scanner generator that we are aware of: `plex`.

1 Motivation

The argumentation for developing *Elex(Prolog)* consists of three steps:

1. *Tokenization* is a separate programming task (separate from reading input on the one hand, and parsing on the other hand).

2. *Scanner generators* are useful to create such tokenizers automatically on the basis of a number of regular expressions.

3. *Multilingual* scanner generators have a number of advantages over language specific tools.

In many (Prolog) programs, the problem arises to analyse input in some format. In some cases, the input may be specified as Prolog terms in which case the Prolog built-in `read` can be used. In most cases, however, the format is different. In such cases, it is important to distinguish between the following sub-tasks. These distinctions are sometimes blurred in actual implementations:

1. *reading* a sequence of characters into a list of character codes

2. *tokenizing* such a list of character codes into a list of tokens

3. *parsing* the list of tokens

Reading vs. tokenizing. The reason for distinguishing reading and tokenizing (contrary to the approach illustrated in chapter 10 of [4]) is the observation that there can be several ways in which input is provided to a program. As an example consider a program which uses socket-IO to obtain its input. For development purposes it is useful if this program can also run in a mode such that it uses standard input in an interactive way. A further disadvantage for integrating reading and tokenizing is that such an integrated approach must take extreme care that in the context of errors in the tokenizer the input channel is properly reset. Similar problems occur if the tokenizer uses backtracking, whereas using backtracking in a scanner which works on a list of character codes is of course trivial.

Note that the size of the input is not an important practical issue anymore for most applications. For example, in SICStus Prolog we have successfully created lists of megabytes of character codes without any problems whatsoever.

Tokenizing vs. parsing. We also distinguish tokenization and parsing. Even if this distinction is commonly accepted in other areas of computer science, it seems that practical Prolog implementations sometimes neglect the distinction. In such implementations, Definite Clause Grammars are used directly on list of character codes. The DCG then contains special clauses to deal with lexical issues such as the recognition of real numbers. Such an approach might complicate the DCG considerably, resulting in increased costs of development and maintenance.

Automatic construction of tokenizers. Not only do we argue for considering tokenization as a separate task, we also argue that there are good reasons to implement such a tokenizer by means of a specialized program which constructs such a tokenizer on the basis of a number of regular expressions. The best-known example of such a program (often called *scanner generator*) is `lex` [2] [1]. Such a scanner generator takes as its input a number of regular expressions (one for each lexical category). Its output consists of a program which implements the tokenizer. In the case of *Elex(Prolog)*, this program defines the predicate `tokenize/2` which can be used for tokenizing a list of character codes into a list of tokens. The use of such a tool can be motivated because:

Declarativeness It can be argued that a regular expression defining, for instance, floating point numbers is more declarative and easier to understand then a DCG on character codes defining the same floating point numbers.

Efficiency If a scanner generator is used, the result is guaranteed to be efficient, by exploiting standard determinization techniques. This efficiency is much harder to obtain 'by hand'.

Flexibility The use of a scanner generator allows for much more flexibility. A change of the scanner only requires a change in a regular expression. For DCG, in contrast, such a change may be more intricate, especially if the DCG was written in a careful way to ensure efficiency.

Ease of Development The declarativeness and flexibility of scanner generators imply that programs can be more easily developed and maintained.

Multilingual Scanner Generators. Well-known examples of scanner generators are `lex` and `flex` which construct output in C. Prolog programmers may be familiar with `plex` [6], which is a scanner generator written in Prolog which provides output in Prolog. *Elex* is a program similar to these, but it extends existing scanner generators because *Elex* aims at providing multiple output languages. The advantages of a single tool for multiple languages are obvious:

- Programmers which use a variety of languages will only need to learn a single scanner generator tool, rather than a number of such tools for each language they use.

- In larger systems modules may be implemented using a variety of programming languages. But these modules may very well need similar scanners. For example, in case these modules communicate by means of some interface language, then each module requires a scanner for this interface language. A multilingual scanner generator will not only reduce the amount of work required to construct these scanners, but, more importantly, it will also immediately guarantee consistency.

Currently, *Elex* is under development, and the number of languages is still rather limited. Recently, we have added Prolog support to *Elex*. In this paper we describe how *Elex* can be used to produce Prolog output, and we describe some issues concerning the implementation of *Elex(Prolog)*.

2 Functionality

The functionality of the *Elex* scanner generator for Prolog is introduced by way of example. Details concerning the functionality of Elex are given in the Elex documentation [5]. In figure 1 we provide a simple example of a scanner definition in *Elex* format. This scanner defines the lexical categories `number` and `word`. It provides output code in Prolog (the parts written between <prolog and >) and C++ (the parts written between <cpp and >).

The first line of the file assigns a name to the scanner (after the keyword `scanner`). This name is used as the name of the module in which the scanner will be defined. The section after the keyword `symbols` is not used for Prolog (it defines a number of symbols to be included in the header of the C++ output file). The section after the keyword `defines` consists of a number of abbreviations. Regular expressions which occur frequently below can be defined here, and given a name. The name can then be used in other definitions instead of the full regular expression. In the example, the abbreviation `number` is defined as a sequence of one or more occurrences of the characters 0..9. The section between the keywords `begin` and `end` consists of a number of rules. Each rule consists of:

- the keyword `on`
- the name of the rule
- a regular expression
- a code fragment for each language

In the case of Prolog, the code fragment will be used as the body of a clause in the resulting scanner. Three special variables can be used in the Prolog fragments. `_Token` is the sequence of character codes which matches the regular expression. `_Tokens0` and `_Tokens` are used as a difference list representation of the resulting list of tokens.

As a prototypical example, consider the rule for `Number`. If the scanner indeed found a sequence of character codes matching the regular expression, then `_Token` will be bound to the list of these character codes. The predicate `number_chars` (built-in in e.g. SICStus Prolog) will convert this list to a Prolog number. The token which is unified with the first element of _Tokens0 is a unary term indicating the nature of the token (`number` in this case) and its actual value.

```
scanner text
symbols
  Number Word
defines
  number = "0-9+"
begin
  on Number "<number>(\.<number>)?([eE][\+\-]?<number>)?"
  <prolog
    number_chars(Nr,__Token), __Tokens0=[number(Nr)|__Tokens].
  >
  <cpp
        return Number;
  >
  on Word "[a-zA-Z][a-zA-Z\-]*"
  <prolog
    atom_chars(Word,__Token), __Tokens0=[w(Word)|__Tokens].
  >
  <cpp
        return Word;
  >
  on WhiteSpace "[ \t\n]+"
  <prolog
    __Tokens0=__Tokens.
  >
  <cpp
    return SymNULL;
  >
  on error
  <prolog
    __Tokens = [skip(__Char)|Ts], tokenize(__Chars,Ts).
  >
  <cpp
    return SymERROR;
  >
end
```

Figure 1: Simple example of an *Elex* script.

The reason that we manipulate the difference list encoding of the resulting list of tokens explicitly is that it provides for greater flexibility. This way, it is easy to assign multiple tokens for a single regular expression, or to skip material completely. An example of the latter is provided by the `WhiteSpace` rule. This rule matches a sequence of one or more occurrences of space, tab and newline. Because the variables `__Tokens0` and __Tokens are unified in the code part, this implies that such white space is ignored in the result.

The `error` rule is special. The rule is applied when a position in the sequence of character codes is encountered for which no regular expression is applicable. In this case the special variables are different too, namely `__Char`, `__Chars`, and __Tokens, such that at the time the error occurs, `__Char` is bound to the first character of the remaining list of character codes, `__Chars` are the rest of the remaining characters, and `__Tokens` is the list of tokens, to be associated with these remaining character codes. In the example, a lexical category `skip(__Char)` is assigned to the first character; the predicate `tokenize/2` is used to tokenize the remaining characters (note that this predicate is the actual result of the scanner generator). Obviously, other definitions of the error rule can be given in order to abort computation, or to raise an exception, or to silently ignore this character and tokenize the remaining ones.

3 Implementation

Introduction. The implementation of the *Elex* scanner generator consists of two parts: automaton construction and code generation. The first part constructs on the basis of an *Elex* script a deterministic finite automaton, in which each final state of the automaton is associated with the name of a rule. In the case a final state can be reached for more than a single rule, the order of the rules in the input file determines which rule has priority: rules which are defined further down in the script have preference (note that this is opposite to the convention implemented in `lex`). The second part generates the actual code for such a given finite automaton. This second part is different for each output language. Therefore, in order to add Prolog support for *Elex* we only need to worry about this second part.

Even if the finite automaton which is constructed on the basis of the regular expressions is determinized, this does not mean that the actual problem of tokenizing a given string is deterministic, since the choice between on the one hand accepting a token (and continuing to find the next token) and on the other hand continuing the current token is often not deterministic. To take an example from the *Elex* documentation, consider the regular expression `for|forest|end`. If the input is the string `forend` then at the time the scanner is considering the 'e' symbol, it doesn't know whether to accept `for` as a token, and to use 'e' as the first character of the next token, or whether the scanner should try to find `forest`.

A simple backtracking scheme is used in such cases. *Elex* provides an option which indicates the number of *tokens* of lookahead the scanner should use. A larger number naturally results in a slower, but possibly more useful, scanner. The implementation of this backtracking scheme should be taken care of in the code generation part.

A few remaining non-deterministic choices remain. For instance, consider the regular expression `for|forest|est`. For the input `forest` two analyses are available. In such cases the scanner operates in a greedy way: longer matches have preference. For this example this implies that only a single token will be returned. Again, this preference for longest matches should be implemented in the code generation part.

```
%% tokenize(+Chars,-Tokens)
tokenize([],[]).
tokenize([Char|Chars0],Tokens0) :-
    (  tokenize(Tokens0,Tokens1,[Char|Chars0],Chars1,Goal),
       look_ahead(Chars1,<LOOKAHEAD>)
    -> call(Goal), tokenize(Chars1,Tokens1)
    ;  error(Char,Chars0,Tokens0)
    ).

look_ahead([],_).
look_ahead([H|T],N) :- look_aheadX(N,[H|T]).

look_aheadX(0,_) :- !.
look_aheadX(N0,[H|T]) :-
    tokenize(_,_,[H|T],Str,_), N is N0-1, look_ahead(Str,N).
```

Figure 2: Generic clauses of the Prolog scanner

Automaton. Let the result of compiling the regular expressions of an *Elex* script be a deterministic finite state automaton $\langle Q, q_0, \Sigma, \delta, F\rangle$, such that:

- Q is the set of states.
- $q_0 \in Q$ is the start state.
- Σ is the set of symbols. We assume Σ is the set of integers $0 \ldots 255$.
- δ is the (partial) transition function mapping $Q \times \Sigma$ to Q
- $F \subseteq Q$ is the set of final states

Furthermore, such a finite automaton is associated with T, τ, π such that:

- T is the set of tokens (the names of the rules used in the *Elex* script.
- τ is a function from F to T (assigning a token to a final state).
- π is a function defined on T which returns the corresponding Prolog fragment for each token.

Code generation. The resulting Prolog tokenizer will extend the set of clauses given in figure 2. Here, the expression `<LOOKAHEAD>` is used as a meta variable which will be replaced in the actual code by an integer ≥ 0 indicating the number of tokens of look ahead.

The if-then-else construct in the second clause of the `tokenize/2` predicate takes care of look-ahead and error handling. Moreover, given the order of the clauses to be defined below, such that clauses corresponding to transitions will precede clauses corresponding to final states, this construct also implements the longest match preference (cf. section 4 for a discussion of the implementation of look-ahead).

The remaining work is encoded by a number of clauses defining the predicate `tokenize/5`. These clauses are defined as follows:

1. For start state q_0 there is a clause:

```
tokenize(Ts0,Ts,S0,S,G):-
    tokenizeq0(Ts0,Ts,S0,S,T,[],T,G).
```

2. For each state $q \in Q$ such that there is a $\sigma \in \Sigma$ such that $\delta(q, \sigma)$ is defined, there is a clause:

```
tokenizeq(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    tokenizeq(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
```

3. For each $\delta(q_i, \sigma) = q_j$) there is a clause

```
tokenizeqi(σ,Ts0,Ts,S0,S,Sy0,Sy,T,G):-
    tokenizeqj(Ts0,Ts,S0,S,Sy0,Sy,T,G).
```

4. For each $\tau(q) = N$ there is a clause

```
tokenizeq(Ts0,Ts,S,S,Sy,Sy,T,N(T,Ts0,Ts)).
```

5. Finally, for each $N \in T$ there is a clause

```
N(_Token,_Tokens0,_Tokens):-π(N)
```

The ordering of the resulting clauses is not important, except that clauses introduced in (2) should precede clauses of the same predicate introduced in (4). This requirement enforces the preference for longer matches.

Example. As an example, consider the tiny *Elex* script for the 'forend' example mentioned previously. This script is given in figure 3. The clauses needed to complete figure 2 for this example are given in figure 5. The intermediate finite automaton is given in figure 4.

Compact scanners. The resulting Prolog code is efficient in that it exploits first argument indexing. However, the resulting code can get rather verbose. For this reason *Elex(Prolog)* can also create a somewhat less efficient but much more compact program. Instead of providing a clause for each state and each symbol, the compact version combines all transitions for a state into a single clause. Arithmetic tests are used to descriminate between cases. Note that in some compilers, such as in newer versions of SICStus Prolog, such tests in the `if` part of an `if-then-else` do not create a choice point. For such compilers the compact version is not much slower than the version given above (and loading the tokenizer is much faster). We do not formally define the compact implementation, but merely refer to the example in the appendix where we provide the resulting *compact* tokenizer for the example in figure 1.

In order to provide an indication of the trade-offs involved, we compared a relatively simple tokenizer in compact and expanded form. The expanded version took about 5.9 CPU-seconds to tokenize a 4 MB list of character codes (size of tokenizer about 1400 lines of Prolog code). The compact version consists of about 400 lines, and takes about 8.6 seconds for tokenization (using SICStus Prolog `3 #6`). [1] For SWI-Prolog (Version 2.8.6), the differences were larger: about 14.7 versus about 32.9 CPU-seconds.

[1] CPU-time was measured on a HP 9000/780 machine running HP-UX 10.20.

```
scanner tw
begin
  on Forend "for|end|forend"
  <prolog
    atom_chars(W,__Token), __Tokens0=[w(W)|__Tokens].
  >
  on WhiteSpace "[ \t\n]+"
  <prolog
    __Tokens0=__Tokens.
  >
  on error
  <prolog
    __Tokens = [skip(__Char)|Ts], tokenize(__Chars,Ts).
  >
end
```

Figure 3: *Elex* script for the 'forend' example.

4 Look Ahead

The implementation of look ahead given in the previous section is simple, straightforward and flexible, but has a number of drawbacks (if look ahead is greater than 0):

- the implementation is inefficient since sequences of characters are scanned more than once (during look ahead and during the actual recognition).
- look ahead is performed even in cases where no ambiguity arises.
- the number of tokens of look-ahead needs to be specified by the programmer.

A more efficient implementation of look-ahead would complicate both the code generation phase and the resulting Prolog code considerably. We argue that a more fruitful approach would be to solve the problem in the automaton construction part as follows. Each rule R in the *Elex* script can be considered as defining a transduction from the language defined by the regular expression r to the name of that specific rule: $r \times R$. The transduction specified by an *Elex* script can then be described as the Kleene closure of the union of each of these transductions: $(r_1 \times R_1 | r_2 \times R_2 \ldots)*$. The resulting transducer can then be subject to a determinization algorithm for transducers [3], [7]. In cases where the resulting transducer is not determinizable a number of static criteria must be defined which forces determinizability (such as a static counterpart of the longest match preference). If this proposal can be implemented, then each of the drawbacks listed above disappear. As a result, both the code generation part and the resulting code can remain both simple, straightforward and efficient. An implementation of this proposal is foreseen for the next release of *Elex*.

5 Comparison with `plex`

The only other scanner generator which produces output in Prolog that we are aware of is `plex` by Peter Reintjes. An important advantage of *Elex* over `plex` is the fact that *Elex* is multilingual. Arguments in favor of multilingual scanner generators have been given in the introduction.

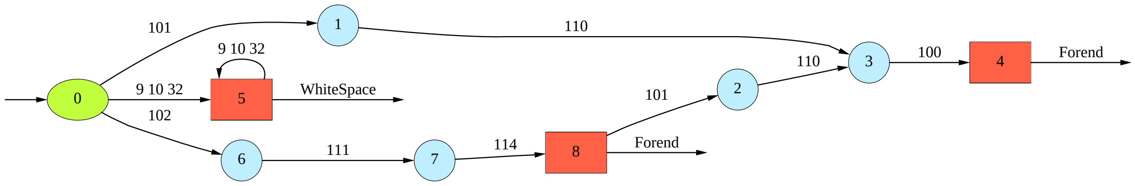

Figure 4: Finite automaton corresponding to the ‘forend’ example.

```
tokenize(Ts0,Ts,S0,S,G):- tokenize0(Ts0,Ts,S0,S,T,[],T,G).

tokenize0(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize0(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize1(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize1(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize2(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize2(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize3(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize3(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize5(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize5(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize6(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize6(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize7(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize7(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize8(Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):- tokenize8(C,Ts0,Ts,S0,S,Sy0,Sy,T,G).

tokenize0(  9,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize0( 10,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize0( 32,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize0(101,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize1(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize0(102,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize6(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize1(110,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize3(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize2(110,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize3(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize3(100,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize4(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize5(  9,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize5( 10,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize5( 32,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize5(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize6(111,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize7(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize7(114,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize8(Ts0,Ts,S0,S,Sy0,Sy,T,G).
tokenize8(101,Ts0,Ts,S0,S,Sy0,Sy,T,G):- tokenize2(Ts0,Ts,S0,S,Sy0,Sy,T,G).

tokenize4(Ts0,Ts,S,S,Sy,Sy,T,'Forend'(T,Ts0,Ts)).
tokenize5(Ts0,Ts,S,S,Sy,Sy,T,'WhiteSpace'(T,Ts0,Ts)).
tokenize8(Ts0,Ts,S,S,Sy,Sy,T,'Forend'(T,Ts0,Ts)).

'Forend'(__Token,__Tokens0,__Tokens):-
    atom_chars(W,__Token), __Tokens0=[w(W)|__Tokens].

'WhiteSpace'(__Token,__Tokens0,__Tokens):-
    __Tokens0=__Tokens.

error(__Char,__Chars,__Tokens) :-
    __Tokens = [skip(__Char)|Ts], tokenize(__Chars,Ts).
```

Figure 5: Specific clauses of the Prolog tokenizer for the ‘forend’ example.

A further striking difference with `plex` is the fact that *plex* not only produces Prolog code, but is implemented in Prolog as well. The installation requires Quintus Prolog. In contrast, *Elex* requires Perl5 and C++ (both of which are available for free).[2] Related to this may be the observation that scanner generation in `plex` is rather slow; the documentation mentions scanner generation times of several minutes. In contrast, *Elex* requires only a few seconds for all examples we tried (including some scripts we use for practical programming work).

The `plex` documentation is not entirely clear about how conflicts in the scanner are solved, and how non-determinism is treated. It seems that `plex` implements in a deterministic way a look-ahead of a single token. In this sense, the resulting Prolog code might be more efficient (and more complicated) then our simple implementation of look-ahead given above, but less flexible (it cannot treat cases which require a larger look-ahead). A newer version of *Elex* in which non-determinism is treated in the automaton construction part will be both efficient, simple and flexible.

The code produced by `plex` uses a similar technique for indexing as we implemented for *Elex(Prolog)*. A compact version, as we provide too, is mentioned in [6] as an interesting alternative which is not part of `plex`.

Two further arguments are given in favor of `plex` in [6]: the possibility to call other tokenizers recursively, and the possibility to construct tokenizers on the fly. The latter possibility is of course not provided in *Elex(Prolog)* simply because the language in which *Elex* is implemented is different from the output language. The first possibility is available in *Elex(Prolog)* too by using the module prefixes.

Availability

Elex sources are made available under GNU Public License restrictions. The *Elex* homepage is `http://www.ozemail.com.au/~mpp/elex/elex.html`. A version of *Elex* including Prolog support is available from `http://www.let.rug.nl/~vannoord/Elex/index.html` under the same conditions. *Elex* is implemented in Perl5 and `C++` and will work on most UNIX platforms. The resulting Prolog code is compatible with at least SICStus Prolog, SWI-Prolog, BProlog, BinProlog and presumably Quintus Prolog.

Acknowledgements

I am grateful to Matthew Phillips for constructing *Elex* and for useful feedback during the development of *Elex(Prolog)*.

References

[1] S.C. Johnson and M.E. Lesk. Language development tools. *Bell System Technical Journal*, 57(6), 1978.

[2] M.E. Lesk and E. Schmidt. Lex – a Lexical Analyzer Generator. Technical report, Bell Laboratories, 1975. CS Technical Report No. 39.

[2] Since the author does not have Quintus Prolog, the following remarks are solely based on [6] and the `plex` documentation.

[3] Mehryar Mohri. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2:61–80, 1996. Originally appeared in 1994 as Technical Report, institut Gaspard Monge, Paris.

[4] Richard A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.

[5] Matthew Phillips. Elex user's guide, 1997. Available as `http://www.ozemail.com.au/~mpp/elex/elex.html`.

[6] Peter B. Reintjes. MULTI/PLEX: An AI Tool for Formal Languages, 1994.

[7] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2), 1995.

Example of Prolog scanner generated by *Elex*

The following lists the resulting Prolog program for the example presented in figure 1. In this case, *Elex(Prolog)* was instructed to produce *compact* code.

```
:- module( text, [ tokenize/2 ]).

tokenize([],[]).
tokenize([Char|Chars0],Tokens0) :-
    (   tokenize(Tokens0,Tokens1,[Char|Chars0],Chars1,Goal),
        look_ahead(Chars1,0)
    ->  call(Goal), tokenize(Chars1,Tokens1)
    ;   error(Char,Chars0,Tokens0)
    ).

look_ahead([],_).
look_ahead([H|T],N) :-  look_aheadX(N,[H|T]).

look_aheadX(0,_) :- !.
look_aheadX(N0,[H|T]) :-
    tokenize(_,_,[H|T],Str,_), N is N0-1, look_ahead(Str,N).

tokenize(Tokens0,Tokens,String0,String,Goal):-
    tokenize(0,Tokens0,Tokens,String0,String,Symbol,[],Symbol,Goal).

tokenize(0,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C >= 9, C =< 10   -> tokenize(8,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 32          -> tokenize(8,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 48, C =< 57  -> tokenize(1,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 65, C =< 90  -> tokenize(7,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 97, C =< 122 -> tokenize(7,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(1,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C =:= 46          -> tokenize(2,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 48, C =< 57  -> tokenize(1,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 69          -> tokenize(4,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 101         -> tokenize(4,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(2,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C >= 48, C =< 57  -> tokenize(3,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(3,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
```

```
    (   C >= 48, C =< 57  -> tokenize(3,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 69          -> tokenize(4,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 101         -> tokenize(4,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(4,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C =:= 43          -> tokenize(5,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 45          -> tokenize(5,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 48, C =< 57  -> tokenize(6,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(5,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C >= 48, C =< 57  -> tokenize(6,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(6,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C >= 48, C =< 57  -> tokenize(6,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(7,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C =:= 45 -> tokenize(7,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 65, C =< 90  -> tokenize(7,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C >= 97, C =< 122 -> tokenize(7,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(8,Ts0,Ts,[C|S0],S,[C|Sy0],Sy,T,G):-
    (   C >= 9, C =< 10  -> tokenize(8,Ts0,Ts,S0,S,Sy0,Sy,T,G)
    ;   C =:= 32         -> tokenize(8,Ts0,Ts,S0,S,Sy0,Sy,T,G)).

tokenize(1,Ts0,Ts,S,S,Sy,Sy,T,'Number'(T,Ts0,Ts)).
tokenize(3,Ts0,Ts,S,S,Sy,Sy,T,'Number'(T,Ts0,Ts)).
tokenize(6,Ts0,Ts,S,S,Sy,Sy,T,'Number'(T,Ts0,Ts)).
tokenize(7,Ts0,Ts,S,S,Sy,Sy,T,'Word'(T,Ts0,Ts)).
tokenize(8,Ts0,Ts,S,S,Sy,Sy,T,'WhiteSpace'(T,Ts0,Ts)).

'Number'(__Token,__Tokens0,__Tokens):-
    number_chars(Number,__Token),
    __Tokens0=[number(Number)|__Tokens].

'Word'(__Token,__Tokens0,__Tokens):-
    atom_chars(Word,__Token),
    __Tokens0=[w(Word)|__Tokens].

'WhiteSpace'(__Token,__Tokens0,__Tokens):-
    __Tokens0=__Tokens.

error(__Char,__Chars,__Tokens) :-
    __Tokens = [skip(__Char)|TokensRest],
    tokenize(__Chars,TokensRest).
```