AST generator specification

Evgenii Balai

September 25, 2014

Contents

1 Vocabulary

- *identifier* - a sequence of alphanumeric or underscore characters starting with a letter.
- *numeral* - a sequence of characters used to represent a number. Numerals can be in one of the following forms:

  1. $d_1 \dots d_k$
  2. $d_1 \dots d_n.d_1 \dots d_m$

  possible preceded by a minus sign, where each d_i is a digit, $k > 0, n \geq 0$ and $m > 0$.

2 Input Specification

The input consists of the following:

1. lexicon file (defined in section 2.1)

2. grammar file (defined in section 2.2)

3. source file (defined in section 2.3)

2.1 Lexicon file

A lexicon file is an ASCII file that contains a specification of types of terminal symbols (*lexems*) that may occur in a program file. Each lexem type is declared as

$$lexem\_type\_name = regex \tag{1}$$

where $lexem\_type\_name$ is an identifier and $regex$ is a regular expression following python syntax[1] whose special characters are limited to ., $*$, $+$, ?, $\{m\}$, $\{m, n\}$, [],\ or |.

We will say that statement (1) *defines* a lexem type named $lexem\_type\_name$.

Lexicon file may contain multiple declarations of the form (1), one per line, where no lexem type name occurs in the left hand side of a statement more than once, and no lexem type name is a member of the set $\{id, num\}$.

2.2 Grammar File

A grammar file is an ASCII file that contains a specification of non-terminals occurring in the produced abstract syntax tree as well as the desired structure of the tree. Every grammar file must be associated with a lexicon file. We will refer to the set of lexem type names defined in the file as S_L.

Non-terminals are specified by statements of the form

$$nt[\mathscr{L}] = \alpha_1\ \alpha_2\ \ldots\ \alpha_n \tag{2}$$

where

1. nt is an identifier which is not a member of the set $\{id, num\} \cup S_L$;

2. each α_i is an identifier;

3. $\mathscr{L}$ is a space-separated sequence of the form $id\ \alpha_{k_1}\ \ldots\ \alpha_{k_m}$ where id is an identifier and each α_{k_i} is an element of the sequence $\alpha_1 \ldots \alpha_n$ on the right hand side of (2).

A grammar file may contain a sequence of statements of the form (2) such that

1. each α_i is in one of the forms b or b_r where

   - b is a name of a non-terminal whose name appears in the left hand side of another statement of the form (2), or is an element of the set of identifiers $\{id, num\} \cup S_L$
   - r is a natural number in the range $1..n$

2. if α_i and α_j are two different elements of the sequence on the right hand side of (2), then

   - at least one of them is of the form b_r, where r is a natural number in the range $1..n$

- if α_i is of the form b_r1 and α_j is of the form b_r2, where $r1$ and $r2$ are two natural numbers in the range $1..n$, then $r1 \neq r2$.

3. if there are two statements

$$nt^1[\mathscr{L}] = \alpha_1^1\ \alpha_2^1\ \ldots\ \alpha_{n_1}^1$$

and

$$nt^2[\mathscr{L}] = \alpha_1^2\ \alpha_2^2\ \ldots\ \alpha_{n_2}^2$$

in the grammar file, then nt^1 is not of the form nt^2_r, where r is a natural number.

2.3 Source File

A source file is a file containing arbitrary collection of ASCII characters.

2.4 Input Example

In this example we will define Algebraic Chess Notation [2] by means of lexicon and grammar defined in sections 2.1 and 2.2 and give an example of a game described in this notation.

2.4.1 Lexicon file

```
figure = K|Q|R|B|N
file = [a-h]
rank = [1-8]
cell = [a-h][1-8]
capture_char = x
space = \s
spaces = \s+
dot = \.
en_passant = e\.p\.
natural_number = [1-9][0-9]+
move_id = [1-9][0-9]+\.
long_castling = 0-0-0
short_castling = 0-0
plus = \+
pound_sign = #
game_over = 1-0|1/2-1/2|0-1
```

2.4.2 Grammar file

```
game[game move_d] = move_d
game[game move_d game] = move_d  spaces game
move_d[move move_id move_1 move_2] = move_id space move_1 move_2
move_d[game_over move move_id move_1] = move_id space move_1 game_over
move[pawn_move cell] = cell
move[move figure_spec cell] = figure_spec cell
move[capture figure_spec cell] = figure_spec capture_char cell
move[pawn_capture file cell] = file capture_char cell
```

```
move[pawn_special_capture] = file capture_char cell en_passant
move[promotion cell figure] = cell figure
move[castling long_castling] = long_castling
move[castling short_castling] = short_castling
check[check move] = move plus
checkmate[checkmate move] = move pound_sign
figure_spec[fig figure] = figure
figure_spec[fig figure file] = figure file
figure_spec[fig figure file] = figure rank
figure_spec[fig figure cell] = figure cell
```

2.4.3 Source File

```
1. e4 e5 2. Qh5 Nc6 3. Bc4 Nf6 4. Qxf7#
```

3 Output specification

The output is obtained in two steps

1. *Lexing. Lexer Module* takes a source file and lexicon file and outputs a sequence of annotated lexemes as specified in section 3.1.
2. *Parsing. Parser Module* takes an output of the lexer module and grammar file as an input and returns an abstract tree as specified in section 3.2.

3.1 Lexing

In addition to thee set of lexeme types declared in the lexicon file, denoted by S_L, we introduce two more lexemes: *id* and *num*. For each lexeme l, we define a regular expression $\mathscr{R}_L(l)$ as follows:

1. if $l \in S_L$, $\mathscr{R}_L(l) = expr$, where $expr$ is the regular expression on the right hand side of the statement
$$l = expr$$
appearing in the lexicon file;
2. if l is *id*, $\mathscr{R}_L(l)$ is
$$[a-z][a-z\_]+$$
3. if l is *num*, $\mathscr{R}_L(l)$ is
$$-?[1-9][0-9]+|-?0\backslash.[0-9]+|-?[1-9][0-9]+\backslash.[0-9]+$$

We will say that a string S *matches* a regular expression E if S is a member of the set of strings specified by the python regular expression

$$\wedge E\$$$

In other words, S matches E if the following python code prints `True`, when being run on a python3.4 interpreter:

```
import re
regex = re.compile(r"^E$")
print(regex.match("S") != None)
```

The symbols $\wedge$ and \$ are added to E to ensure that the whole string, but not its prefix or suffix are matched. More details on the syntax and semantics of python regular expressions can be found in [1].

Let I be the string that represents the contents of the input file. The *lexing sequence* of I with respect to the lexicon file L is a sequence of pairs $(l_1, s_1), \ldots (l_n, s_n)$, such that:

1. each l_i is a lexeme which is a member of the set $S_L \cup \{id, num\}$;
2. each s_i is a string;
3. $s_1 + \ldots + s_n = I$ (where $+$ denotes concatenation);
4. for each $1 \leq i \leq n$, s_i matches $\mathscr{R}_L(l_i)$;
5. for each $1 \leq i \leq n$, s_i is the longest prefix of $s_i + \ldots + s_n$ such that s_i matches $\mathscr{R}_L(l)$ for some $l \in S_L \cup \{id, num\}$

We will refer to each member of the lexing sequence as an *annotated lexeme*.

3.2 Parsing

Given a sequence of annotated lexemes $(l_1, s_1), \ldots, (l_n, s_n)$ and a grammar file F, the goal of parsing is to produce an abstract syntax tree as defined in this section.

Let G be the sequence of statements in F. By G_{BNF} we denote a BNF grammar obtained from G by the following steps:

1. For each statement of the form 2 in G, add new statements

$$nt_{k_1}[\mathscr{L}] = \alpha_1 \ \alpha_2 \ \ldots \ \alpha_n$$

$\ldots$

$$nt_{k_n}[\mathscr{L}] = \alpha_1 \ \alpha_2 \ \ldots \ \alpha_n$$

   where each of $nt_{k_1}, ..., nt_{k_n}$ is an element of a sequence on the right hand side of a statement in G.

2. Remove $[\mathscr{L}]$ from each statement of the form 2.

Let $S_0 \to S_1 \to \ldots \to S_n$ be a leftmost derivation of $l_1 \ldots l_n$ in G_{BNF}, where S_0 is the non-terminal occurring in the left hand side of the first statement in G. Let

$$nt[id \ \alpha_{k_1} \ \alpha_{k_2} \ \ldots \ \alpha_{k_m}] = \alpha_1 \ \alpha_2 \ \ldots \ \alpha_n$$

be the statement in G corresponding to the production rule of G_{BNF} that can be used to derive S_1 from S_0.

The *abstract syntax tree* corresponding to the derivation is a list defined as follows:

1. the first element of the list is id
2. for each $1 \leq i \leq m$, the $(i+1)^{th}$ element of the list is an abstract syntax tree defined as follows:
   - if α_i is a terminal in G, and (α_i, s) is a member of the lexing sequence, the abstract syntax tree is the list whose only element is (α_i, s)
   - if α_i is a non-terminal, the abstract syntax tree corresponds to the left-most derivation $S_b \rightarrow \ldots \rightarrow S_e$, where S_{b+1} is obtained from S_b by expanding α_i and S_e is the first element of $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ the non-terminals only from $\alpha_{i+1}, \ldots, \alpha_n$.

3.3 Example

The lexing sequence for the lexicon file given in section 2.1 and source file given in section 2.3 is:

```
(move_id, '1.'), (space, ' '), (cell, 'e4'), (space, ' '),
(cell, 'e5'), (spaces, ' '), (move_id, '2.'), (space, ' '),
(figure, 'Q'), (cell, 'h5'), (space, ' '), (figure, 'B'),
(cell, 'c6'), (spaces, ' '), (move_id, '3.'), (space, ' '),
(figure, 'B'), (cell, 'c4'), (space, ' '),  (figure, 'N'),
(cell, 'f6'), (spaces, ' '), (move_id, '4.'), (space, ' '),
(figure, 'Q'), (capture_char, 'x'), (cell, 'f7'),
(pound_sign, '#')
```

Given a grammar file in section 2.2, the corresponding abstract syntax tree is:

```
[ game,
      [move, [(move_id '1.')], [pawn_move [(cell 'e4')] ],
                               [pawn_move [(cell 'e5')] ] ],
      [game,
           [move, [(move_id '2.')], [move [fig [(figure 'Q')]]
                                    [(cell 'h5')] ],
                                    [move, [fig [(figure 'N')]],
                                    [(cell 'c6')]]],
           [game,
               [move, [(move_id '3.')], [move [fig [(figure 'B')]],
                                              [(cell 'c4')] ],
                                        [move, [fig [(figure 'N') ]],
                                        [(cell 'f6')]]
               ],
               [game,
                       [checkmate   [move, [(move_id '4.')],
                                       [capture [fig [(figure 'Q')] ],
                                       [(cell 'f7')] ] ] ]
               ]
           ]

      ]
]
```

References

[1] Python Software Foundation. Regular expression operations – python 3.4.1 documentation. https://docs.python.org/3.4/library/re.html.

[2] Wikipedia. Algebraic notation (chess). http://en.wikipedia.org/wiki/Algebraic_notation_%28chess%29.