AST generator specification

Evgenii Balai

September 29, 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. $d_1 \ldots d_k$
  2. $d_1 \ldots d_n.d_1 \ldots d_m$

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

$$nt(\mathscr{L}) = \alpha_1(\tau_1)\ \alpha_2(\tau_2)\ \ldots\ \alpha_n(\tau_n) \tag{2}$$

where

1. nt is an identifier which is not a member of the set $\{id, num\} \cup S_L$ sometimes referred to as a *non-terminal name*;
2. each α_i is an identifier;
3. each τ_i is an identifier such that if $i \neq j$, then $\tau_i \neq \tau_j$;
4. $\mathscr{L}$ is a space-separated sequence of the form $\beta_1\ \ldots\ \beta_m$ where each β_i except possibly β_1 is one of the following forms:
    (a) an identifier that is an element of the sequence $\tau_1 \ldots \tau_n$;

(b) $f(c)$, where f is a member of the set of identifiers $\{cut\_root\}$ and c is an element of the sequence $\tau_1 \dots \tau_n$. If $f = cut\_root$, c must not be a lexeme type name.

If β_1 is of one of the forms (a) - (b), then m must be equal to 1, otherwise β_1 is an identifier.

We allow a shorthand α that stands for $\alpha(\alpha)$ to be an element of the sequence on the right hand side of the statement (2).

A grammar file may contain a sequence of statements of the form (2), one per line, such that each α_i is a non-terminal name occurring on the left hand side of another statement in the sequence.

2.3 Source File

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

2.4 Input Examples

2.4.1 Arithmetic Expression

In this example we will define an arithmetic expression whose operands are integer numbers by means of lexicon and grammar defined in sections 2.1 and 2.2 and give an example of an arithmetic expression that can be used as contents of a source file.

2.4.1.1 Lexicon File

```
num = -?([1-9][0-9]+|0)
add_op = \+|-
mult_op = :|\*
left_paren = \(
right_paren = \)
```

2.4.1.2 Grammar File

```
expr(T1) = T1
T1(T2) = T2
T1(add M1 M2) = T2(M1) add_op T1(M2)
T2(mult Op1 Op2) = T3(Op1) mult_op T2(Op2)
T2(T3) = T3
T3(expr) = left_paren expr right_paren
T3(num) = num
```

2.4.1.3 Source File

```
1+2*3
```

2.4.2 Chess Notation

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.

2.4.2.1 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 = #
end = 1-0|1/2-1/2|0-1
```

2.4.2.2 Grammar file

```
game(game move_d) = move_d
game(game move_d cut_root(G)) = move_d  spaces game(G)
move_d(move move_id M1 M2) = move_id space move(M1) move(M2)
move_d(game_over  move_id move) = move_id space move end
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
```

2.4.2.3 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 sequence G of statements from a grammar file, the goal of parsing is to produce an abstract syntax tree as defined in this section.

By G_{BNF} we denote a BNF grammar obtained from G by replacing each statement in G of the form (2) with a production rule

$$nt = \alpha_1 \ \alpha_2 \ \ldots \ \alpha_n$$

The symbols of G_{BNF} which are lexeme type names in G are all and only terminal symbols of G_{BNF}.
Let $\mathscr{T}$ be a derivation tree for $l_1 \ldots l_n$ in G_{BNF}, where the starting symbol of the derivation is the non-terminal occurring in the left hand side of the first statement in G. We will say that $\mathscr{T}_1$ is a *subtree* of $\mathscr{T}$ if $\mathscr{T}_1$ a derivation tree whose nodes are a node n in $\mathscr{T}$ and all of n's descendants in $\mathscr{T}$ and whose edges are all and only edges of $\mathscr{T}$ connecting a pair of nodes of $\mathscr{T}_1$.

For each node N of $\mathscr{T}$ by R_N we denote a production rule of G_{BNF} that was applied to expand N in $\mathscr{T}$. We will also say that the statement of G

$$nt(\beta_1, \ldots, \beta_m) = \alpha_1(\tau_1) \ \alpha_2(\tau_2) \ \ldots \ \alpha_n(\tau_n)$$

corresponds to N if it was used to obtain R_N.

Let N be a node of $\mathscr{T}$ and, if N is not a leaf node of $\mathscr{T}$, let

$$nt(\beta_1, \ldots, \beta_m) = \alpha_1(\tau_1) \ \alpha_2(\tau_2) \ \ldots \ \alpha_n(\tau_n)$$

be the statement of G corresponding to N.

For a sequence $\beta_{b_1}, \ldots, \beta_{b_t}$ such that $\{\beta_{b_1}, \ldots, \beta_{b_t}\} \subseteq \{\beta_1, \ldots, \beta_m\}$ and each $\beta_{b_i} \in \{\beta_{b_1}, \ldots, \beta_{b_t}\}$ is either a member of $\{\tau_1, \ldots, \tau_n\}$ or is of the form $f(c)$ where c is a member of $\{\tau_1, \ldots, \tau_n\}$, by $A(\tau_{p_1}, \ldots, \tau_{p_t})$ we denote a sequence of derivation trees defined as follows:

1. if $t = 1$ and $\beta_{p_1} = \tau_i$ for some $1 \leq i \leq n$, $A(\tau_{p_1})$ is a sequence containing one element which is a subtree of $\mathscr{T}$ rooted at the i^{th} child of N.

2. if $t = 1$ and $\beta_{p_1} = \mathit{cut\_root}(\tau_i)$ for some $1 \leq i \leq n$, $A(\tau_{p_1})$ is a sequence of trees whose j^{th} element is a subtree of $\mathscr{T}$ rooted at j^{th} child of the i^{th} child of N.

3. if $t > 1$, $A(\tau_{p_1}, \ldots, \tau_{p_t})$ is a sequence obtained by concatenating the sequences $A(\tau_{p_1}), \ldots, A(\tau_{p_t})$.

The *abstract syntax tree corresponding to* N is defined as follows:

1. if N is a leaf of $\mathscr{T}$ labeled with l_i, the abstract syntax tree is a tree whose only node is labeled with the pair (l_i, s_i);

2. if N is not a leaf of $\mathscr{T}$, and $\beta_1 = \tau_i$ for some $1 \leq i \leq n$ the abstract syntax tree is the tree corresponding to the i^{th} child of N (from left to right) in $\mathscr{T}$;

3. if N is not a leaf of $\mathscr{T}$ and $\beta_1 \neq \tau_i$ for all $1 \leq i \leq$, the abstract syntax tree is defined as follows:

(a) the root r of the tree is labeled by β_1;

(b) the children of r(from left to right) are roots of the trees $\mathscr{T}_1, \dots \mathscr{T}_k$, where $k = |A(\beta_2, \dots, \beta_m)|$ and for each $1 \leq i \leq k$, $\mathscr{T}_i$ is an abstract syntax tree corresponding to the i-th element of $A(\beta_2, \dots, \beta_m)$.

The result of the parsing is defined to be the abstract syntax tree corresponding to the root of $\mathscr{T}$.

3.3 Representation of trees as lists

Let T be an abstract syntax tree. In the implementation, T is represented by a list as follows:

1. if T is empty, it is represented by an empty lists
2. it T consists of only one node labeled by r, the corresponding list is $[r]$
3. if a tree consists of more then one node, it is represented by the list $[r, t_1, \dots, t_n]$, where r is a label of the root of T and t_i represents the subtree of T rooted at i^{th} child of T

3.4 Examples

3.4.1 Arithmetic Expression Tree

The lexing sequence for the lexicon file given in section 2.4.1.1 and source file given in section 2.4.1.3 is:

```
(num '1'), (add_op, '+'), (num, '2'), (mult_op,'*'), (num, '3')
```

Given the grammar file in section 2.4.1.2, the corresponding abstract syntax tree is represented by the list:

```
[add, (num,'1'),[mult,(num,'2'),(num,'3')]]
```

3.4.2 Chess Game Tree

The lexing sequence for the lexicon file given in section 2.4.2.1 and source file given in section 2.4.2.3 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 the grammar file in section 2.4.2.2, the corresponding abstract syntax tree is represented by the list:

```
[ game,
      [move, [(move_id '1.')], [pawn_move [(cell 'e4')] ],
                                [pawn_move [(cell 'e5')] ] ],

      [move, [(move_id '2.')], [move [fig [(figure 'Q')]]
                                      [(cell 'h5')] ],
                                      [move, [fig [(figure 'N')]],
                                      [(cell 'c6')]]],

      [move, [(move_id '3.')], [move [fig [(figure 'B')]],
                                                [(cell 'c4')] ],
                                          [move, [fig [(figure 'N') ]],
                                          [(cell 'f6')]]],

      [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.