AST generator specification

Evgenii Balai

October 2, 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. $d_1 \ldots d_n.d_1 \ldots d_m$

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 $\{id, num, space\}$.

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

$$nt(\beta_1 \ \ldots \ \beta_m) = \alpha_1(\tau_1) \ \ldots \ \alpha_n(\tau_n) \tag{2}$$

where

1. nt is an identifier which is not a member of the set $\{id, num, space\} \cup S_L$ sometimes referred to as a *non-terminal symbol*, or simply *non-terminal*;
2. $n \geq 1$;
3. each α_i is an identifier;
4. each τ_i is an identifier;
5. $\beta_1 \ \ldots \ \beta_m$ $(m \geq 1)$ is a sequence of distinct identifiers, such that each β_i except possibly β_1 is in one of the following forms:

(a) an identifier that is an element of the sequence $\tau_1 \dots \tau_n$;

(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;

6. if β_i is of one of the forms τ_j or $f(\tau_j)$, then τ_j must occur in the sequence $\tau_1 \dots \tau_n$ exactly once;

7. if β_1 is of one of the forms τ_j or $f(\tau_j)$, then m must be equal to 1, otherwise β_1 is an identifier also referred to as *a label*.

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).

We will refer to the sequence of statements of the form(2) as *a grammar*, if each α_i is a non-terminal name occurring on the left hand side of another statement in the sequence, or a member of the set$\{id, num, space\} \cup S_L$. The non-terminal symbols occurring in the statements of the grammar are referred to as the *non-terminal symbols* (or the *non-terminals*) *of the grammar*. The non-terminal symbol occurring on the left hand side of the first statement of G is referred to as the *starting symbol of the grammar*. The members of S_L are referred to as the *terminal symbols* (or the *terminals*) *of the grammar*. Both terminals and non-terminals of a grammar are referred to as *symbols of the grammar*. Finally, the labels occurring in the statements of a grammar are referred to as the *labels of the grammar*.

Let G be a grammar. We define a leveling function $||$ of G that maps non-terminals of G onto a set of non-negative integers in the range [0..n]. A leveling function is called reasonable if for each statement of the form

$$nt(\beta_1 \ \dots \ \beta_m) = \alpha_1(\tau_1)$$

such that α_1 is a non-terminal, $|nt| > |\alpha_1|$. G is called *stratified*, if there exists a reasonable leveling of G.

A grammar file contains a stratified grammar, one statement per line.

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
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*, *num* and *spaces*. 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
$$-?[1-9][0-9]+|-?0\backslash.[0-9]+|-?[1-9][0-9]+\backslash.[0-9]+$$
4. if l is *spaces*, $\mathscr{R}_L(l)$ is
$$\backslash s+$$

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 element 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 G the goal of parsing is to produce an abstract syntax tree as defined in this section.

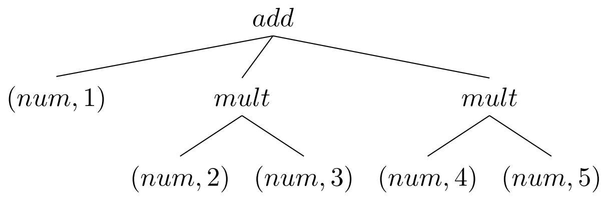
3.3 Abstract Syntax Trees for a Grammar

We say that T is *an abstract syntax tree for a grammar G* if T is a finite rooted ordered[1] tree whose non-leaf nodes are labeled by labels of G and whose leafs are labeled by terminals of G.

Let T_1 and T_2 be two trees of G. We say T_1 is a *subtree* of T_2 if the following three conditions hold:

1. the root of T_2 is a node of T_1;
2. the nodes of T_2 different from r are the descendants of r in T_1;
3. every edge of T_2 is an edge of T_1.

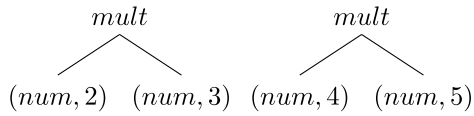
Example 1. *Consider the tree T shown below*[2] *:*



[1] A finite tree is *ordered* if the children of each node of the tree are pairwise distinct and totally ordered. That is, we can define a correspondence between the children of a node V of the tree and the set of natural numbers $\{1, \ldots, \ldots, n\}$, and refer to the children of V as $1^{st}, 2^{nd}, \ldots, n^{th}$ child of V.

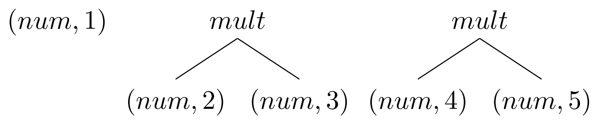
[2] The children of nodes in the tree are shown from left to right according to their order.

The tree T has 8 subtrees: T itself, the five trees containing only one node which is a leaf of T and two trees rooted at the nodes labeled by mult shown below:



If T is an abstract tree of G, by *cut_root* we define a sequence of trees i-th of which is rooted at i-th child of the root of T.

Example 2. *If T is a tree given in example 1, cut_root is a sequence of three trees shown below (from left to right):*



3.4 Parsing output specification

Let G be a stratified grammar with a symbol S, $(l_1, s_1), \ldots, (l_n, s_n)$ be a lexing sequence denoted by L, b and e be two natural numbers such that $1 \leq b \leq e \leq n$.

By $R_{G,L}(S, b, e)$ we denote an abstract syntax tree for G satisfying one of the following two conditions:

1. if $b = e$ and S is l_b, and there is a statement in G of the form

$$S(\tau_1) = l_b(\tau_1)$$

   $R_{G,L}(S, b, e)$ contains only one node labeled by (l_b, s_b);

2. if S is a non-terminal of G, and there is a statement in G of the form

$$S(\beta_1 \ldots \beta_m) = \alpha_1(\tau_1) \ldots \alpha_n(\tau_n)$$

   where

   - $l_b \ldots l_e = l^1_1, l^1_2 \ldots l^1_{k_1} \ldots l^n_1 \ldots l^n_{k_n}$ for some $1 \leq k_1, \ldots, k_n \leq (e-b+1)$,
   - for each $1 \leq i \leq n$ there exists a tree denoted by

$$R_{G,L}(\alpha_i, k_1 + \ldots + k_{i-1} + 1, k_1 + \ldots + k_i)$$

   then

   - if $\beta_1 = \tau_j$ for some $1 \leq j \leq n$, then
     $R_{G,L}(S, b, e)$ is $R_{G,L}(\alpha_j, k_1 + \ldots + k_{j-1} + 1, k_1 + \ldots + k_j)$
   - if $\beta_1 \neq \tau_j$ and $\beta_1 \neq cut\_root(\tau_j)$ for all $1 \leq j \leq n$,
     then $R_{G,L}(S, b, e)$ is a tree whose root r is labeled by β_1 and the tree T_i whose root is the i^{th} child of r is obtained as follows:
     (a) Let $f : \{2..m\} \rightarrow \{1..n\}$ be a function such that for any $j \in \{2..m\}$ $\beta_j = \tau_{f(j)}$ or $\beta_j = cut\_root(\tau_{f(j)})$.

(b) For each $1 \leq j \leq n$ let A_j be an abstract syntax tree denoted by $R(\alpha_j, k_1 + \ldots + k_{j-1} + 1, k_1 + \ldots + k_j)$

(c) Let $c : \{2..m\} \to \mathbb{N}$ be a function defined as follows:

$$c(u) = \begin{cases} 1, & \beta_u \in \{\tau_1 \ldots \tau_n\} \\ \text{the number of children of the root of } T_{f(u)}, & \text{otherwise} \end{cases}$$

(d) Let p be the largest number in the range $[1..m]$ such that

$$c(2) + \ldots + c(p) < i$$

(e) if β_{p+1} is of the form $cut\_root(\tau_j)$, T_i is the $(i-(c(2)+\ldots+c(p)))^{th}$ child of A_p; otherwise T_i is A_p

We will refer $R_{G,L}(S, b, e)$ as an abstract syntax tree of G *matching the lexing sequence* $l_b, \ldots, l_e$ *on* S. The result of the parsing is defined to be the abstract syntax tree matching the lexing sequence $l_b, \ldots, l_e$ on the starting symbol of G.

3.5 Trees Representation

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

1. if T is empty, it is represented by an empty list;
2. it T consists of only one node labeled by a pair (l_i, s_i), it is represented the pair (l_i, s_i);
3. if a tree consists of more then one node, it is represented by the list $[r, t_1, \ldots, t_n]$, where r is a label of the root of T and t_i is the representation of the subtree of T rooted at i^{th} child of T

3.6 Examples

3.6.1 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')]]
```

3.6.2 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.