Semantic Web — Spring 2009
Homework #1

Simple XML Parser using Python Lex/Yacc

Behnam Esfahbod
behnam@sharif.edu

April 14, 2009

Listing 1: The Python program

```
#!/usr/bin/env python

import sys
from UserString import UserString

from ply import lex, yacc



# #######
# Debug
#

DEBUG = {
    'INPUT': False,
    'TOKENS': False,
    'PARSER': False,
    'OUTPUT': False,
}

def _debug_header(part):
    if DEBUG[part]:
        print '--------'
        print '%s:' % part

def _debug_footer(part):
    if DEBUG[part]:
        pass

def _debug_print_(part, s):
    if DEBUG[part]:
        print s



# ########
# TOKENS

tokens = [

    # INITIAL
    'CDATA',
    'OPENTAGOPEN',
    'CLOSETAGOPEN',

    # tag
    'TAGATTRNAME',

    'TAGCLOSE',
    'LONETAGCLOSE',

    'ATTRASSIGN',

    # attrvalue1
    'ATTRVALUE1OPEN',
    'ATTRVALUE1STRING',
    'ATTRVALUE1CLOSE',

    # attrvalue2
    'ATTRVALUE2OPEN',
    'ATTRVALUE2STRING',
    'ATTRVALUE2CLOSE',

]

# Regulare expressions

re_digit       = r'([0-9])'
re_nondigit    = r'([_A-Za-z])'
re_identifier  = r'(' + re_nondigit + r'(' + re_digit + r'|' + re_nondigit + r')*)'

class SyntaxError(Exception):
    pass



class XmlLexer:
    # The XML Tokenizer

    # states:
    #
    #   default:
    #     The default context, non-tag texts
    #   tag:
    #     A document tag
    #   string:
    #     Within quote-delimited strings inside tags

    states = (
        ('tag', 'exclusive'),
        ('attrvalue1', 'exclusive'),
        ('attrvalue2', 'exclusive'),
    )

    tokens = tokens


    # ANY

    def t_ANY_error(self, t):
        raise SyntaxError("Illegal character '%s'" % t.value[0])
        t.lexer.skip(1)
        pass


    # INITIAL

    t_ignore  = ''
    def t_CLOSETAGOPEN(self, t):
        r'</'
        t.lexer.push_state('tag')
        return t

    def t_OPENTAGOPEN(self, t):
        r'<'
        t.lexer.push_state('tag')
        return t

    def t_CDATA(self, t):
        '[^<]+'
        return t


    # tag: name

    t_tag_ignore  = ' \t'

    def t_tag_TAGATTRNAME(self, t):
        return t
    t_tag_TAGATTRNAME.__doc__ = re_identifier

    def t_tag_TAGCLOSE(self, t):
        r'>'
        t.lexer.pop_state()
        return t

    def t_tag_LONETAGCLOSE(self, t):
        r'/>'
        t.lexer.pop_state()
        return t


    # tag: attr

    t_tag_ATTRASSIGN     = r'='

    def t_tag_ATTRVALUE1OPEN(self, t):
        r'\''
        t.lexer.push_state('attrvalue1')
        return t

    def t_tag_ATTRVALUE2OPEN(self, t):
        r'"'
        t.lexer.push_state('attrvalue2')
        return t


    # attrvalue1

    def t_attrvalue1_ATTRVALUE1STRING(self, t):
        r'[^\']+'
        t.value = unicode(t.value)
        return t

    def t_attrvalue1_ATTRVALUE1CLOSE(self, t):
        r'\''
        t.lexer.pop_state()
        return t

    t_attrvalue1_ignore  = ''


    # attrvalue2

    def t_attrvalue2_ATTRVALUE2STRING(self, t):
        r'[^"]+'
        t.value = unicode(t.value)
        return t

    def t_attrvalue2_ATTRVALUE2CLOSE(self, t):
        r'"'
        t.lexer.pop_state()
        return t

    t_attrvalue2_ignore  = ''


    # misc

    literals = '$%^'

    def t_ANY_newline(self, t):
        r'\n'
        t.lexer.lineno += len(t.value)


    # Build the lexer
    def build(self, **kwargs):
        self.lexer = lex.lex(object=self, **kwargs)

    # Test it output
    def test(self, data):
        self.lexer.input(data)

        _debug_header('TOKENS')

        while 1:
            tok = self.lexer.token()
            if not tok: break
            _debug_print_('TOKENS', '[%-12s] %s' % (self.lexer.lexstate, tok))

        _debug_footer('TOKENS')

    # XmlLexer ends


# ########
# Escape

_xml_escape_table = {
    "&": "&amp;",
    '"': "&quot;",
    "'": "&apos;",
    ">": "&gt;",
    "<": "&lt;",
    }

def _xml_escape(text):
    L=[]
    for c in text:
        L.append(_xml_escape_table.get(c,c))
    return "".join(L)

def _xml_unescape(s):
    rules = _xml_escape_table.items()
    rules.reverse()

    for x, y in rules:
        s = s.replace(y, x)

    return s


# ########
# PARSER

tag_stack = []

# Customization

def parser_trace(x):
    _debug_print_('PARSER', '[%-16s] %s' % (sys._getframe(1).f_code.co_name, x))

def yacc_production_str(p):
    #return "YaccProduction(%s, %s)" % (str(p.slice), str(p.stack))
    return "YaccP%s" % (str([i.value for i in p.slice]))

yacc.YaccProduction.__str__ = yacc_production_str

class ParserError(Exception):
    pass

# Grammer

def p_root_element(p):
    '''
    root : element
    root : element CDATA
    '''
    parser_trace(p)

    p[0] = p[1]

def p_root_cdata_element(p):
    '''
    root : CDATA element
    root : CDATA element CDATA
    '''
    parser_trace(p)

    p[0] = p[2]

def p_element(p):
    '''
    element : opentag children closetag
    element : lonetag
    '''
    parser_trace(p)

    if len(p) == 4:
        p[1].children = p[2]

    p[0] = p[1]

# tag

def p_opentag(p):
    'opentag : OPENTAGOPEN TAGATTRNAME attributes TAGCLOSE'
    parser_trace(p)

    tag_stack.append(p[2])
    p[0] = DOM.Element(p[2], p[3])

def p_closetag(p):
    'closetag : CLOSETAGOPEN TAGATTRNAME TAGCLOSE'
    parser_trace(p)

    n = tag_stack.pop()
    if p[2] != n:
        raise ParserError('Close tag name ("%s") does not match the corresponding
            open tag ("%s").' % (p[2], n))
```

```
def p_lonetag(p):
    'lonetag : OPENTAGOPEN TAGATTRNAME attributes LONETAGCLOSE'
    parser_trace(p)

    p[0] = DOM.Element(p[2], p[3])

# attr

def p_attributes(p):
    '''
    attributes : attribute attributes
    attributes : nothing
    '''
    parser_trace(p)

    if len(p) == 3:
        if p[2]:
            p[1].update(p[2])
            p[0] = p[1]
        else:
            p[0] = p[1]
    else:
        p[0] = {}

def p_attribute(p):
    'attribute : TAGATTRNAME ATTRASSIGN attrvalue'
    parser_trace(p)

    p[0] = {p[1]: p[3]}

def p_attrvalue(p):
    '''
    attrvalue : ATTRVALUE1OPEN ATTRVALUE1STRING ATTRVALUE1CLOSE
    attrvalue : ATTRVALUE2OPEN ATTRVALUE2STRING ATTRVALUE2CLOSE
    '''
    parser_trace(p)

    p[0] = _xml_unescape(p[2])

# child

def p_children(p):
    '''
    children : child children
    children : nothing
    '''
    parser_trace(p)

    if len(p) > 2:
        if p[2]:
            p[0] = [p[1]] + p[2]
        else:
            p[0] = [p[1]]
    else:
        p[0] = []

def p_child_element(p):
    'child : element'
    parser_trace(p)

    p[0] = p[1]

def p_child_cdata(p):
    'child : CDATA'
    parser_trace(p)

    p[0] = DOM.Text(p[1])
```

```
# nothing

def p_nothing(p):
    'nothing :'
    pass

# Error rule for syntax errors
def p_error(p):
    raise ParserError("Parse error: %s" % (p,))
    pass


# ########
# DOM

class DOM:
    class Element:
        # Document object model
        #
        # Parser returns the root element of the XML document

        def __init__(self, name, attributes={}, children=[]):
            self.name = name
            self.attributes = attributes
            self.children = children

        def __str__(self):
            attributes_str = ''
            for attr in self.attributes:
                attributes_str += ' %s="%s"' % (attr, _xml_escape(self.attributes[
                    attr]))

            children_str = ''
            for child in self.children:
                if isinstance(child, self.__class__):
                    children_str += str(child)
                else:
                    children_str += child

            return '<%s%s>%s</%s>'% (self.name, attributes_str, children_str, self.
                name)

        def __repr__(self):
            return str(self)

    class Text(UserString):
        pass

# ########
# MAIN

def parse(data):

    _debug_header('INPUT')
    _debug_print_('INPUT', data)
    _debug_footer('INPUT')

    # Tokenizer
    xml_lexer = XmlLexer()
    xml_lexer.build()

    xml_lexer.test(data)

    # Parser
    yacc.yacc(method="SLR")

    _debug_header('PARSER')
    root = yacc.parse(data)
    _debug_footer('PARSER')

    _debug_header('OUTPUT')
    _debug_print_('OUTPUT', root)
    _debug_footer('OUTPUT')

    return root


def tree(node, level=0, prefix=''):
    'Returns a tree view of the XML data'

    s_node = prefix + node.name + ':'

    s_children = ''

    children = node.children
    children.reverse()

    if len(children) == 1 and not 'name' in children[0].__dict__:
        s_node += ' %s' % node.children[0] + '\n'

    else:
        first = True
        for i in xrange(len(children)):
            if 'name' in node.children[i].__dict__:
                p = '    '
                s_children = tree(node.children[i], level+1, prefix+p) + s_children

        s_node += '\n'

    return s_node + s_children

def main():

    data = open(sys.argv[1]).read()
    root = parse(data)
    print tree(root)


if __name__ == '__main__':
    main()
```