LL(1) leftmost derivation and LR(1) rightmost derivations.

CSC 310, Spring, 2009, handout for March 9 class and midterm study guide
Dr. Dale E. Parson, http://faculty.kutztown.edu/parson
Midterm is on Wednesday, March 11

This study guide uses two grammars defined in directory ~parson/ProcLang/ll1_lr1/ to illustrate a top-down, leftmost derivation and a bottom-up, rightmost derivation of a right-recursive LL(1) grammar, and to illustrate a bottom-up, rightmost derivation of a left-recursive LR(1) grammar that accepts the same strings as the right-recursive, LL(1) grammar. Note that any LL(1) grammar is also LR(1) — a LR(1) parser can parse LL(1) grammars — but in this case right-recursion in the LL(1) grammar leads to an inefficient LR(1) parse.

Here are the relevant test cases and grammar files from ~parson/ProcLang/ll1_lr1/.

`gmake testll1` runs a right-recursive LL(1) grammar parser defined in file ll1.g. This grammar uses the YAPPS2 LL(1) parser generator that we are using for assignment 2.

`gmake testlr1r` runs the same right-recursive LL(1) grammar (which is always LR(1) as well) parser defined in file lr1_rightrecursion.py. This grammar uses the PLY LALR(1) parser generator from http://www.dabeaz.com/ply/, which was written by our Python textbook author.

`gmake testlr1l` runs a left-recursive LR(1) grammar parser defined in file lr1_leftrecursion.py that accepts the same input sentences as the right-recursive, LL(1) grammar. This grammar uses the PLY LALR(1) parser generators well. It is not LL(1) because of the left recursion. It is more efficient for bottom-up, LR(1) parsing than the left-recursive grammar in lr1_rightrecursion.py.

Both grammars accept zero or more even digits (from the set [02468]) followed by zero or more odd digits (from the set [13579]). We could match these strings with a DFA, but I am using these simple grammars in order to be able to walk through example parsings within a class period. These grammars illustrate the points without generating an over-abundance of text.

Below are example debug traces of these parsers, parsing the string 000111. We will go over these derivations in detail on March 9.
Leftmost derivation of a parse of 000111 using the LL(1) grammar in ll1.g.

The screen dump uses «SUBSCRIPTS» to distinguish each top-down expansion of a nonterminal to a sentential form. The nonterminal is to the left of the “->” symbol, and the right side of a production about to be parsed is on the right side. Thus “program -> evens odds” appears as “program<<0>> -> evens<<1>> odds<<1>> END,” and you can find the rightmost derivation by first expanding “evens<<1>>” to “0 evens<<2>>” in the top-down order of the print statements below. Note the right recursion of the evens and odds nonterminal productions. We will step through this derivation on March 9.

1 LL1(1) Test driver for evenodd parser using right recursion.
2 The grammar is this:
3
4     program -> evens odds
5
6     evens   -> EVEN evens
7             | ‘’
8
9     odds   -> ODD odds
10             | ‘’
11
12 >>> LL1 INPUT STRING: 0001111
13
14     program<<0>> -> evens<<1>> odds<<1>> END
15     evens<<1>> -> 0 evens<<2>>
16     evens<<2>> -> 0 evens<<3>>
17     evens<<3>> -> 0 evens<<4>>
18     evens<<4>> -> ‘’
19     odds<<1>> -> 1 odds<<2>>
20     odds<<2>> -> 1 odds<<3>>
21     odds<<3>> -> 1 odds<<4>>
22     odds<<4>> -> 1 odds<<5>>
23     odds<<5>> -> ‘’
24     0 0 0 ‘’ 1 1 1 1 ‘’
25 >>>
Rightmost derivation of a parse of 000111 using the same LL(1) grammar (that is also LR(1)) in lr1_rightrecursion.py.

This debug trace of a PLY parse of the same grammar as the previous example shows the rightmost derivation. Note that, since we are scanning from the left in an LR(1) parse, reductions of stack states happen left-to-right. The “R” in “LR(1)” denotes the fact that, in deriving the sentence 000111 from the grammar, you perform the formal derivation from the top. In other words, a bottom-up parser matches a rightmost derivation in a series of backward steps. Read the parse trace from the last line to the first line in order to see the rightmost derivation from the nonterminal program that defines this grammar. We will walk through this example March 9.

Note at line 34 it is necessary to shift all contiguous EVEN tokens (000 in this example) onto the stack before performing a series of reductions initiated by the epsilon production for nonterminal evens. Line 80 shows a similar inefficiency for nonterminal odds. This inefficient accumulation of data on the stack comes from the right-recursive nature of this example. Rewriting the grammar as a left-recursive grammar in the following example corrects the problem of memory consumption.

1 LALR(1) Test driver for evenodd parser using right recursion.
2 The grammar is this:
3 4 program -> evens odds
5 6 evens -> EVEN evens
7 | ‘’
8 9 odds -> ODD odds
10 | ‘’
11 12 >>> LALR1 INPUT STRING: 0001111
13 14 PLY: PARSE DEBUG START
15 16 State : 0
17 Stack : LexToken(EVEN,’0’,1,0)
18 Action : Shift and goto state 1
19 20 State : 1
21 Stack : EVEN . LexToken(EVEN,’0’,1,1)
22 Action : Shift and goto state 1
23 24 State : 1
25 Stack : EVEN EVEN . LexToken(EVEN,’0’,1,2)
26 Action : Shift and goto state 1
27 28 State : 1
29 Stack : EVEN EVEN EVEN . LexToken(ODD,’1’,1,3)
30  Action : Reduce rule [epsilon -> <empty>] with [] and goto state 6
31  Result : <NoneType @ 0x7fb32474> (None)
32
33  State : 2
34  Stack : EVEN EVEN EVEN epsilon . LexToken(ODD,’1’,1,3)
35  Action : Reduce rule [evens -> epsilon] with [None] and goto state 3
36  Result : <NoneType @ 0x7fb32474> (None)
37
38  State : 5
39  Stack : EVEN EVEN EVEN evens . LexToken(ODD,’1’,1,3)
40  Action : Reduce rule [evens -> EVEN evens] with [‘0’,None] and goto state 2
41  Result : <NoneType @ 0x7fb32474> (None)
42
43  State : 5
44  Stack : EVEN EVEN evens . LexToken(ODD,’1’,1,3)
45  Action : Reduce rule [evens -> EVEN evens] with [‘0’,None] and goto state 2
46  Result : <NoneType @ 0x7fb32474> (None)
47
48  State : 5
49  Stack : EVEN evens . LexToken(ODD,’1’,1,3)
50  Action : Reduce rule [evens -> EVEN evens] with [‘0’,None] and goto state 2
51  Result : <NoneType @ 0x7fb32474> (None)
52
53  State : 3
54  Stack : evens . LexToken(ODD,’1’,1,3)
55  Action : Shift and goto state 7
56
57  State : 7
58  Stack : evens ODD . LexToken(ODD,’1’,1,4)
59  Action : Shift and goto state 7
60
61  State : 7
62  Stack : evens ODD ODD . LexToken(ODD,’1’,1,5)
63  Action : Shift and goto state 7
64
65  State : 7
66  Stack : evens ODD ODD ODD . LexToken(ODD,’1’,1,6)
67  Action : Shift and goto state 7
68
69  State : 7
70  Stack : evens ODD ODD ODD ODD . Send
71  Action : Reduce rule [epsilon -> <empty>] with [] and goto state 6
72  Result : <NoneType @ 0x7fb32474> (None)
73
74  State : 6
75  Stack : evens ODD ODD ODD ODD epsilon . Send
Action : Reduce rule [odds -> epsilon] with [None] and goto state 5
Result : <NoneType @ 0x7fb32474> (None)

State : 9
Stack : evens ODD ODD ODD ODD odds . $end
Action : Reduce rule [odds -> ODD odds] with [‘1’,None] and goto state 4
Result : <NoneType @ 0x7fb32474> (None)

State : 9
Stack : evens ODD ODD ODD odds . $end
Action : Reduce rule [odds -> ODD odds] with [‘1’,None] and goto state 4
Result : <NoneType @ 0x7fb32474> (None)

State : 9
Stack : evens ODD ODD ODD odds . $end
Action : Reduce rule [odds -> ODD odds] with [‘1’,None] and goto state 4
Result : <NoneType @ 0x7fb32474> (None)

State : 9
Stack : evens ODD ODD odds . $end
Action : Reduce rule [program -> evens odds] with [None,None] and goto state 1
Result : <NoneType @ 0x7fb32474> (None)

State : 4
Stack : program . $end
Done : Returning <NoneType @ 0x7fb32474> (None)
PLY: PARSE DEBUG END
>>>
Rightmost derivation of a parse of 000111 using the LR(1) grammar in lr1_leftrecursion.py.

Here is a bottom-up, LR(1) parse of a left-recursive grammar. This grammar eliminates the previous grammar’s over-consumption of stack space while accepting the same sentences. This is not an LL(1) grammar because of the left recursion. Note that the stack states are reduced to nonterminals as soon as possible, thanks to the left recursion.

To see the rightmost derivation, read from the last step to the first steps as with the previous LR(1) parse. Bottom up parses start at the bottom, but the program nonterminal that starts any derivation is at the top.

```
LALR(1) Test driver for evenodd parser using left recursion.
The grammar is this:

program -> evens odds
evens   -> evens EVEN
         | '
odds   -> odds ODD
         | '

>>> LALR1 INPUT STRING: 0001111
PLY: PARSE DEBUG START
State : 0
Stack : . LexToken(EVEN,'0',1,0)
Action : Reduce rule [epsilon -> <empty>] with [] and goto state 6
Result : <NoneType @ 0x1e1a2e40> (None)

State : 1
Stack : epsilon . LexToken(EVEN,'0',1,0)
Action : Reduce rule [evens -> epsilon] with [None] and goto state 3
Result : <NoneType @ 0x1e1a2e40> (None)

State : 2
Stack : evens . LexToken(EVEN,'0',1,0)
Action : Shift and goto state 4

State : 4
Stack : evens EVEN . LexToken(EVEN,'0',1,1)
Action : Reduce rule [evens -> evens EVEN] with [None,'0'] and goto state 2
Result : <NoneType @ 0x1e1a2e40> (None)

State : 2
```
Stack : evens . LexToken(EVEN,'0',1,1)
Action : Shift and goto state 4
State : 4
Stack : evens EVEN . LexToken(EVEN,'0',1,2)
Action : Reduce rule [evens -> evens EVEN] with [None,'0'] and goto state 2
Result : <NoneType @ 0x1e1a2e40> (None)
State : 2
Stack : evens . LexToken(EVEN,'0',1,2)
Action : Shift and goto state 4
State : 4
Stack : evens EVEN . LexToken(ODD,'1',1,3)
Action : Reduce rule [evens -> evens EVEN] with [None,'0'] and goto state 2
Result : <NoneType @ 0x1e1a2e40> (None)
State : 2
Stack : evens . LexToken(ODD,'1',1,3)
Action : Reduce rule [epsilon -> <empty>] with [] and goto state 6
Result : <NoneType @ 0x1e1a2e40> (None)
State : 5
Stack : evens epsilon . LexToken(ODD,'1',1,3)
Action : Reduce rule [odds -> epsilon] with [None] and goto state 5
Result : <NoneType @ 0x1e1a2e40> (None)
State : 6
Stack : evens odds . LexToken(ODD,'1',1,3)
Action : Shift and goto state 7
State : 7
Stack : evens odds ODD . LexToken(ODD,'1',1,4)
Action : Reduce rule [odds -> odds ODD] with [None,'1'] and goto state 4
Result : <NoneType @ 0x1e1a2e40> (None)
State : 6
Stack : evens odds . LexToken(ODD,'1',1,4)
Action : Shift and goto state 7
State : 7
Stack : evens odds ODD . LexToken(ODD,'1',1,5)
Action : Reduce rule [odds -> odds ODD] with [None,'1'] and goto state 4
Result : <NoneType @ 0x1e1a2e40> (None)
State : 6
82 Stack : evens odds . LexToken(ODD,'1',1,5)
83 Action : Shift and goto state 7
84
85 State : 7
86 Stack : evens odds ODD . LexToken(ODD,'1',1,6)
87 Action : Reduce rule [odds -> odds ODD] with [None,'1'] and goto state 4
88 Result : <NoneType @ 0x1e1a2e40> (None)
89
90 State : 6
91 Stack : evens odds . LexToken(ODD,'1',1,6)
92 Action : Shift and goto state 7
93
94 State : 7
95 Stack : evens odds ODD . $end
96 Action : Reduce rule [odds -> odds ODD] with [None,'1'] and goto state 4
97 Result : <NoneType @ 0x1e1a2e40> (None)
98
99 State : 6
100 Stack : evens odds . $end
101 Action : Reduce rule [program -> evens odds] with [None,None] and goto state 1
102 Result : <NoneType @ 0x1e1a2e40> (None)
103
104 State : 3
105 Stack : program . $end
106 Done : Returning <NoneType @ 0x1e1a2e40> (None)
107 PLY: PARSE DEBUG END
108 >>>
# ll1.g -- A simple LL(1) grammar for showing a leftmost derivation.
# Use the YAPPS2 LL(1) parser generator on this grammar.
# CSC 310, Dr. D. Parson, Spring, 2009

# This grammar is a right-recursive LL(1) grammar for parsing zero or more even digits followed by zero or more odd digits. It could be written as a regular expression, but I am doing this just to keep a demo of leftmost and rightmost derivations simple.

# THE LL(1) GRAMMAR:

mygrammar = ""
    program -> evens odds
    evens   -> EVEN evens
             | '
    odds   -> ODD odds
            | '
    ""

%%

token END: "$"
token EVEN: r'[02468]'
token ODD: r'[13579]'

rule program:
    {{ print 'program<<0>> -> evens<<1>> odds<<1>> END' }}
    evens<<1>> odds<<1>> END
    {{ return (evens + ' ' + odds) }}

rule evens<<D>>:
    EVEN
    {{ print 'evens<<' + str(D) + '>> -> ' + EVEN + ' evens<<' + str(D+1) + '>>' }}
    evens<<D+1>>
    {{ return (EVEN + ' ' + evens) }}
    |
    {{ print "evens<<" + str(D) + ">> -> "'" }}
    {{ return "'" }}
rule odds<<D>>:
  ODD
  {{ print 'odds<<' + str(D) + '>> -> ' + ODD + ' odds<<' + str(D+1) + '>>' }}
  odds<<D+1>>
  {{ return (ODD + ' ' + odds) }}
  |
  {{ print "odds<<" + str(D) + ">> -> "" }}
  {{ return "" }}

if __name__=='__main__':
  print 'LL1(1) Test driver for evenodd parser using right recursion.'
  print 'The grammar is this:
' + mygrammar
  while 1:
    try: s = raw_input('>>> ')
    except EOFError: break
    if not strip(s): break
    print "LL1 INPUT STRING: ", s, "\n"
    print parse('program', s)
# lr1_rightrecursion.py -- A parser for a right-recursive grammar as
# translated from LL(1) YAPPS2 into LALR(1) PLY as an example of finding the
# rightmost derivation of a right-recursive grammar.
#
# See evenodd.g for an LL(1) parser for this right-recursive grammar.
#
# See lr1_leftrecursion.py for a left-recursive LALR(1) equivalent.
#
# CSC 310, Dr. D. Parson, Spring, 2009

# THE RIGHT-RECURSIVE LL(1) GRAMMAR IS ALSO LR(1):
#
import ply.lex as lex
import ply.yacc as yacc

# SCANNER DATA SETUP

tokens = (
    'EVEN', 'ODD'
)

# Regular expressions for the tokens.

t_EVEN      =       r'[02468]'  
t_ODD       =       r'[13579]'  

# Ignored characters

t_ignore    =       r'\r\n' 

def t_error(t):     # Taken from a Beazley example grammar.
    print "Illegal character '%s'" % t.value[0] 
    t.lexer.skip(1) 

# BUILD THE SCANNER

lex.lex() 

# Create a dictionary of names for storing parse bindings.

names = {} 

# PLY USES A FUNCTION’S DOCUMENTATION STRING AS A GRAMMAR PRODUCTION.
```python
def p_program(p):
    'program : evens odds'
pass

def p_evens(p):
    '''evens : EVEN evens
            |  epsilon
    '''

def p_odds(p):
    '''odds : ODD odds
            |  epsilon
    '''

def p_epsilon(p):
    'epsilon :'
pass

# BUILD THE PARSER

yacc.yacc()

mygrammar = '''
program ->  evens odds

evens   ->  EVEN evens
          | ' '

odds   ->  ODD odds
          | ' '

epsilon
'''

import logging
from sys import stdout, stderr

if __name__=='__main__':
    print 'LALR(1) Test driver for evenodd parser using right recursion.'
    print 'The grammar is this:
' + mygrammar
    while 1:
        try: s = raw_input('>>> ')
        except EOFError: break
        print 'LALR1 INPUT STRING: ', s, ''
        stdout.flush()
        stderr.flush()
yacc.parse(s,debug=logging.DEBUG)
        stderr.flush()
    print """
92    stdout.flush()
93
import ply.lex as lex
import ply.yacc as yacc
# The PLY LALR(1) parser generator, see http://www.dabeaz.com/ply/.

# SCANNER DATA SETUP

tokens = ( 'EVEN', 'ODD' )

# Regular expressions for the tokens.
t_EVEN = r'[02468]'
t_ODD = r'[13579]'

# Ignored characters
t_ignore = " \
\t\r\n"

def t_error(t):
    print "Illegal character '%s'" % t.value[0]
t.lexer.skip(1)

# BUILD THE SCANNER
lex.lex()

# Create a dictionary of names for storing parse bindings.
names = { }

# PLY USES A FUNCTION’S DOCUMENTATION STRING AS A GRAMMAR PRODUCTION.
def p_program(p):
    'program : evens odds'

    pass
```python
def p_evens(p):
    """evens : evens EVEN
    | epsilon
    """

def p_odds(p):
    """odds : odds ODD
    | epsilon
    """

def p_epsilon(p):
    'epsilon :
    pass

# BUILD THE PARSER

yacc.yacc()

mygrammar = ""
program -> evens odds

evens -> evens EVEN
| '

odds -> odds ODD
| '
"

import logging
from sys import stdout, stderr

if __name__=='__main__':
    print 'LALR(1) Test driver for evenodd parser using left recursion.'
    print 'The grammar is this:\n    print "LALR1 INPUT STRING: ", s, "\n"
    stdout.flush()
    yacc.parse(s,debug=logging.DEBUG)
    stderr.flush()
    print ""
    stdout.flush()
```