Teoría de Lenguajes

Trabajo Práctico

Formato PGN

[Event "USA-02.Congress"]
[Site "Cleveland"]
[Date "1871.??.??"]
[Round "1"]
[White "Mackenzie, George Henry"]
[Black "Haughton, WB."]
[Result "1-0"]
[WhiteElo ""]
[BlackElo ""]
[ECO "C29"]

1.e4 e5 2.Nc3 Nf6 3.f4 d6 4.Bc4 exf4 5.Nf3 h6 6.d4 g5 7.e5 Nh7 8.O-O Be7
9.Qd3 Nf8 10.d5 dxe5 11.Nxe5 Bd6 12.Re1 Qe7 13.Nf3 Bc5+ 14.Kh1 Be6 15.dxe6 fxe6
16.Nd5 Qf7 17.Ne5 Qg7 18.Bd2 c6 19.Nc3 h5 20.Ne4 Qxe5 21.Bc3 Qf5 22.Nd6+ 1-0

[Event "USA-02.Congress"]
[Site "Cleveland"]
[Date "1871.??.??"]
[Round "1"]
[White "Harding, H."]
[Black "Mackenzie, George Henry"]
[Result "0-1"]
[WhiteElo ""]
[BlackElo ""]
[ECO "A20"]

1.a3 e5 2.c4 f5 3.Nc3 Nf6 4.e3 c5 5.d3 Nc6 6.Be2 b6 7.f4 exf4 8.exf4 Bd6
9.Nh3 O-O 10.O-O Nd4 11.Bf3 Nxf3+ 12.Rxf3 Bb7 13.Rf2 Ng4 14.Rf1 Qf6 15.Nf2 Nxf2
16.Rxf2 Rae8 17.Bd2 Qd4 18.Nb5 Qxd3 19.Bc3 Qxd1+ 20.Rxd1 Bb8 21.Rxd7 Rf7
22.Rfd2 Bxf4 23.Rxf7 Kxf7 24.Rd7+ Re7 25.Rxe7+ Kxe7 26.Nxa7 g5 27.Nb5 Be3+
28.Kf1 f4 29.Be5 Kd7 30.Bf6 g4 31.Bg5 Bxg2+ 32.Kxg2 f3+ 33.Kg3 Bxg5 34.Nc3 Bh4+ 0-1

Tags

[White “Deep Blue”]

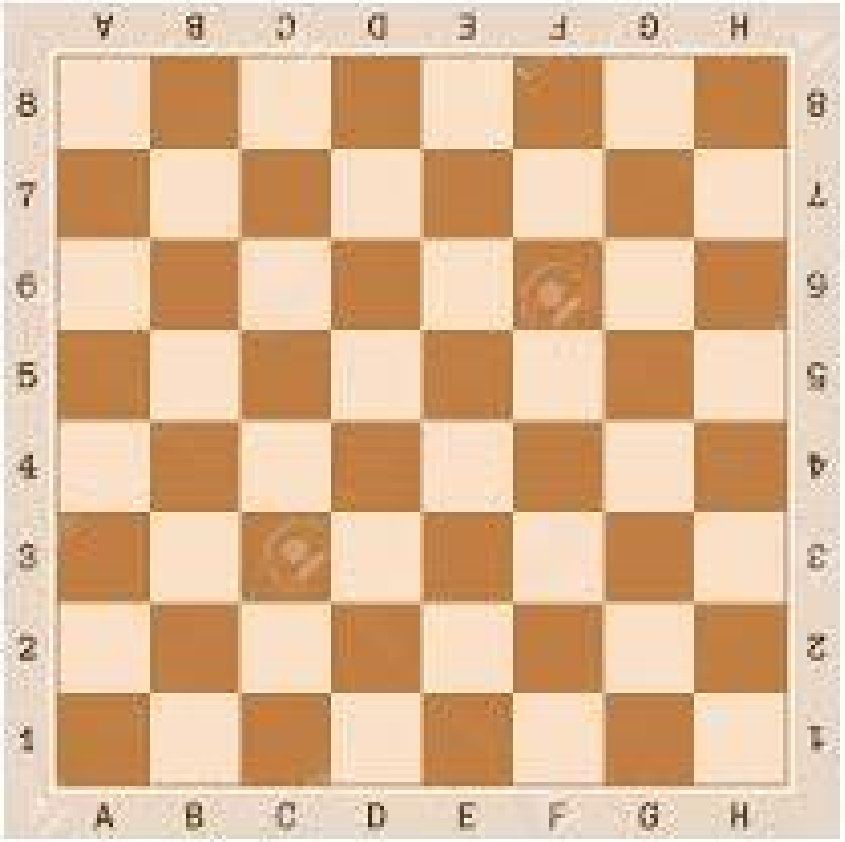
[Black “Garry Kasparov”]

[Date “1997.5.11”]

...

Casilla

a1, a2, ..., a8, b1, ..., h1, ..., h8



Jugada

inicial de pieza (opcional)	P, R, N, B, Q, K
columna (opcional)	a..h
fila (opcional)	1..8
captura (opcional)	x
columna	a..h
fila	1..8
jaque o mate (opcional)	+, #

Jugadas especiales: O-O, O-O-O

(hay más opcional no tenido en cuenta…)

Ejemplo de jugada

B g5

B g5+

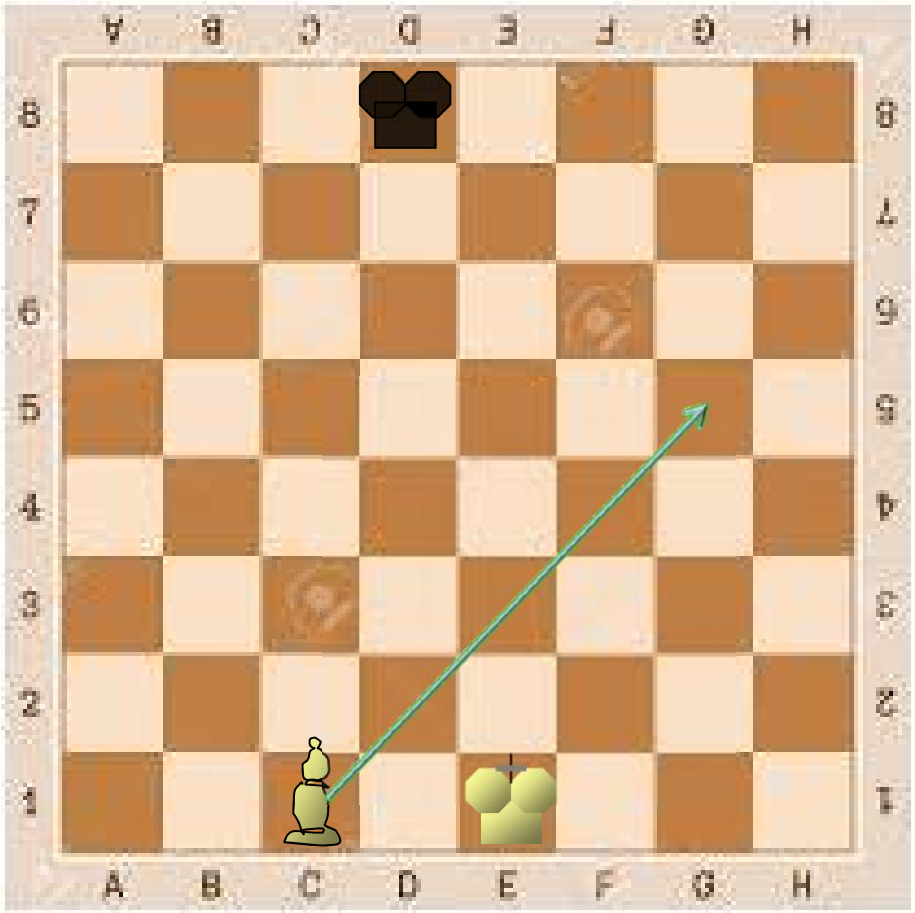
B c g5

B 1 g5

B c1 g5

B x g5

(etc.)



Comentarios

- Encerrados entre { } o ( ).
- Pueden contener jugadas.
- Posibilidad de anidamiento.

Ejemplo:

8... Qe7 9. Nf4 Nf8 10. Qg4 f5 {Es la única posible. No sólo se amenazaba 11. Qxg7 sino también Nxd5.} 11. exf6 gxf6 12. O-O-O {Otra vez amenaza Nxd5.} }

Objetivo

- Validar la sintaxis (superficial) de archivos PGN
- Determinar la jugada 1 (movimiento inicial del blanco) más frecuente en todo el archivo.
- Determinar el máximo nivel de anidamiento de los comentarios que contengan al menos una jugada.

Entrega

- Grupos de 3
- Por e-mail a tptleng@gmail.com
- Archivo zip o rar
- Hasta el 6/8
- Subject: entrega de TP [grupo …, …, …]
  (Ver más detalles en TP.pdf)

Lex

lex.py se usa para “tokenizar” la cadena de entrada

Ejemplo:

x = 3 + 42 * (s - t)

se “tokeniza” a

’x’,’=’, ’3’, ’+’, ’42’, ’*’, ’(’, ’s’, ’-’, ’t’, ’)’

Tokens

```
# Lista de los nombres de los tokens – requerido
 tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
 )
```

Especificación de tokens

```
# Expresiones regulares para tokens
  simples
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
```

Reglas

t_PLUS = r'\+'

Debe usarse algún nombre especificado en la lista de tokens.

Si se necesita una acción, se puede escribir una regla como una función. Este ejemplo reconoce números y convierte la cadena en un entero:

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Palabras reservadas

```
reserved = {
    'if' : 'IF',
    'then' : 'THEN',
    'else' : 'ELSE',
    'while' : 'WHILE',
    ...
 }

 tokens = ['LPAREN','RPAREN',...,'ID'] +
   list(reserved.values())
```

Contar líneas

```
def t_newline(t):
    r'\n+' t.lexer.lineno += len(t.value)
```

Acción

```
# Regla de números dados con una expresión regular + código
 def t_NUMBER(t):
     r'\d+'
     t.value = int(t.value)
     return t

 # Regla que permite contar los números de línea
 def t_newline(t):
     r'\n+'
     t.lexer.lineno += len(t.value)

 # La cadena con los caracteres a ignorar (espacios y tabulados)
 t_ignore  = ' \t'

 # Rutina de manejo de error
 def t_error(t):
     print("Carácter ilegal '%s'" % t.value[0])
     t.lexer.skip(1)

 # Construir el lexer
 lexer = lex.lex()
```

yacc

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

Ejemplo calculado

```
expression0 : expression1 + term     expression0.val = expression1.val + term.val
            | expression1 - term     expression0.val = expression1.val - term.val
            | term                   expression0.val = term.val

term0       : term1 * factor         term0.val = term1.val * factor.val
            | term1 / factor         term0.val = term1.val / factor.val
            | factor                 term0.val = factor.val

factor      : NUMBER                 factor.val = int(NUMBER.lexval)
            | ( expression )         factor.val = expression.val
```

Secuencia

- Símbolos de la gramática como argumento de las funciones.
- Los tipos pueden ser tipos simples, tuplas o instancias…

Ejemplo de regla

```
def p_expression_plus(p):
    'expression : expression PLUS term'
```

p[0] p[1] p[2] p[3]

p[0] = p[1] + p[3]

```
import ply.yacc as yacc

from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Regla de error de sintaxis
def p_error(p):
    print("Error de sintaxis en la entrada")

# Construir el parser
parser = yacc.yacc()

# Ciclo principal
while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Manejo de errores

```
def p_statement_print(p):
    'statement : PRINT expr SEMI'
    ...
```

Agregando la regla:

```
def p_statement_print_error(p):
    'statement : PRINT error SEMI'
    print("Error de sintaxis en print.")
```

se logra la *resincronización*.

Reglas similares

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

⇒

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

Reglas similares

```
def p_expressions(p):
    '''expression : expression MINUS expression
                  | MINUS expression'''
    if (len(p) == 4):
        p[0] = p[1] - p[3]
    elif (len(p) == 3):
        p[0] = -p[2]
```

Literales

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
      term        : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Usar comillas simples. Deben estar declarados en el archivo mandado al lex:

```
literals = ['+','-','*','/']
```

Símbolo inicial

Yacc considera como símbolo inicial el símbolo no terminal izquierdo de la primera regla. De otro modo, este puede especificarse con:

```
start = expression
```

o bien llamando a yacc con

```
parser = yacc.yacc(start='expression')
```

Ambigüedad

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
 )
```

Especifica asociatividad y precedencia.

Se especifican de menor a mayor precedencia.

PLY

https://www.dabeaz.com/ply/ply.html