DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Validación de archivos PGN

Teoría de Lenguajes

Segundo Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Alvarez, Lucas Martín	740/16	luckzoot@gmail.com
Bukovits, Nicolás	546/14	nicko_buk@hotmail.com
Maspi, Gian Agustín	266/19	agustin.maspi@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Resumen

Índice

1. Introducción

En el presente trabajo se buscará analizar un archivo en formato PGN *(Portable Game Notation)*, el cual simula una descripción de movimientos, turnos, anotaciones y puntuaciones de partidas de ajedrez. El objetivo del mismo es, por un lado, verificar si la sintaxis superficial del archivo PGN es válida y, por otro, determinar el máximo nivel de comentarios en que no haya capturas, considerando todas las partidas.

Para esta tarea se necesitará realizar un lexer y un parser, los cuales utilizaremos para analizar el archivo PGN. En primer lugar, se explicará el análisis de la entrada realizada por el lexer. Luego, se desarrollará cómo es la verificación de la sintaxis de la gramática propuesta, tal que la entrada cumpla con las reglas presentadas como partidas válidas de ajedrez (sin tener en cuenta las reglas del juego). Por último, se comentará los atributos utilizados en la semántica y se mostrará ejemplos de funcionamiento del lexer y el parser.

1.1. Herramientas utilizadas

El lexer y el parser fueron implementados en *Python* con la herramienta PLY (*Python Lex-Yacc*), la cual es una implementación de lex y yacc para el lenguaje *Python*.

2. Lexer

El lexer se encargará de descomponer la cadena de entrada en una secuencia de **TOKENS** del lenguaje, los cuales serán los símbolos terminales de la gramática. Al producirse la tokenización, el lexer tendrá diferentes criterios para saber con cual matchear.

1. **En caso de estar definidas como funciones**: La prioridad esta definida por el orden de implementación de las funciones.
2. **En caso que no estén definidas como funciones**: Por orden decreciente de longitud de la expresión regular.

Siempre se tendrá en cuenta el estado en el cual se encuentra el lexer y los estados para los cuales cada función está definida.

En nuestra solución, los tokens propuestos que son generados por el lexer son los siguientes:

- `DESCRIPTOR`: Un corchete de apertura seguido de una cadena arbitraria seguida de otra cadena arbitraria encerrada entre comillas y un corchete de clausura al final. Ninguna de las cadenas pueden contener corchetes de apertura o clausura.
- `RES`: El resultado de la partida. Solo hay tres cadenas posibles:
  1. “1-0” : Indica que ganó el blanco
  2. “0-1” : Indica que ganó el negro
  3. “1/2-1/2”: Indica que hubo un empate *(tablas)*
- `SIMBOLO`: Es una cadena que contiene un símbolo. Indica una característica del movimiento que le precede. Las cadenas posibles son:
  1. “+”: Si la jugada es un jaque.
  2. “++”: Si la jugada es un jaque mate.
  3. “!”: Si la jugada fue considerada como buena.
  4. “?”: Si la jugada fue considerada como mala.
- `MOVIMIENTO`: Es un movimiento de ajedrez válido.

- `LLLAVE`: Inicio de un comentario. Es la cadena "{".
- `RLLAVE`: Fin de un comentario. Es la cadena "}".
- `LPAREN`: Inicio de un comentario. Es la cadena "(".
- `RPAREN`: Fin de un comentario. Es la cadena ")".
- `CONTINUAJUGADA`: Es una cadena que contiene un número seguido de tres puntos. Al tokenizar devolvemos como valor solo al número.
- `NUMBER`: Numeración de la jugada. Es una cadena que contiene un número y un punto. Al tokenizar devolvemos como valor solo al número.
- `TEXTO`: Una cadena de texto arbitraria (no puede incluir espacios). No puede contener llaves o paréntesis de apertura o clausura.

2.1. Dificultades encontradas y decisiones tomadas

Decidimos darle mayor prioridad al token **MOVIMIENTO** que al de **TEXTO**. Esto último tiene como objetivo poder diferenciar, dentro de los posibles comentarios, la aparición de detalles de movimientos o posibles jugadas, las cuales más adelante nos servirán para distinguir los niveles de captura.

Los espacios y las tabulaciones son ignoradas por el lexer. Además, asumimos que si adentro de un comentario hay dos movimientos sin un espacio entre ambos, entonces se trata de dos movimientos independientes.

Para el caso de las jugadas, se decidió no forzar a que haya un espacio entre los movimientos. Por lo tanto, si se ingresan dos movimientos que sin espacio representan uno solo pero con espacio son dos movimientos validos independientes, se mostrará como que se trata de uno solo. La razón de esta decisión es por qué, si no había espacio, el análisis léxico tampoco iba a ser satisfactorio.

Una de las mayores dificultades se presentó al decidir como tokenizar los comentarios: Al usar una expresión regular para el **TEXTO**, esta misma era demasiado general, por lo que capturaba casi toda la cadena de entrada. Para solucionar este problema se introdujeron **estados**.

Por lo tanto, definimos como estado inicial el default *(perteneciente al lexer)* denominado **INITIAL**, el cual representa los tokens que se encuentran fuera de un comentario. Luego, agregamos el estado **insideComment** del tipo exclusivo *(esto significa que las reglas de lexer de este estado reemplazan a las del inicial)*. Este estado nos permite controlar si nos encontramos adentro o afuera de un comentario. Este cometido se logra dado que se cuenta con la variable global **comentarios_abiertos**, la cual vamos incrementando cada vez que se abre un paréntesis o una llave, y se decrementa cada vez que se cierra uno de estos. De esta forma, permitimos al lexer poder separar ambos casos y tratarlos de forma distintiva.

Por lo tanto, cuando se esta adentro de un comentario se utiliza la regla de **TEXTO** para capturar a cualquier cadena (ya que no necesitamos ninguna información especifica sobre ellas, ni diferenciarlas), excepto por los movimientos, los cuales tienen mayor prioridad a la hora de seleccionar la regla del lexer a usar. Por lo tanto, solo hay dos tipos de cadenas que pueden estar adentro de los comentarios: los movimientos y las cadenas arbitrarias sin espacios.

En la Sección 4 se puede observar ejemplos de utilización del lexer y de cadenas válidas que el mismo espera como entrada.

3. Parser

El parser se ocupa de verificar la sintaxis de la gramática, es decir, que el texto obtenido del lexer cumpla con las producciones existentes en la gramática y que, por ende, dicha entrada pertenezca al lenguaje definido por la misma, y tambien de realizar las validaciones semanticas. En este caso estamos lidiando con una gramática **LALR**, para la cual propusimos las siguientes producciones:

```
# Producción inicial: Representa el archivo PGN, con al menos un partido
1. PGN -> HEADER JUGADAS PARTIDA
```

```
# Producción que obliga a que se genere, al menos, una jugada.
2. JUGADAS -> number DESC_JUGADA JUGADA RESULTADO


# Producción que obliga a que se genere, al menos, un descriptor del evento.
3. HEADER -> descriptor DESCRIPTOR


# Producción que genera partidas de ajedrez
4. PARTIDA -> HEADER JUGADAS PARTIDA
            | lambda


# Producción que genera descriptores de evento.
5. DESCRIPTOR -> DESCRIPTOR descriptor
               | lambda


# Producción que genera una serie de jugadas las cuales están numeradas de forma secuencial.
6. JUGADA -> number DESC_JUGADA JUGADA RESULTADO
           | lambda


# Producción para el cuerpo de las jugadas. Se permiten uno o dos movimientos.
7. DESC_JUGADA -> movimiento SIMBOLO COMENTARIO_SIGUIENTE_JUGADA MOVIMIENTO SIMBOLO COMENTARIO


# Producción que genera comentarios que incluyen el número de la jugada al cierre del mismo.
# Indican que la jugada continúa post-comentario.
# En el único lugar donde se pueden generar es inmediatamente después del primer movimiento.
8. COMENTARIO_SIGUIENTE_JUGADA -> lllave CONTENIDO rllave continuajugada
                                | lparen CONTENIDO rparen continuajugada
                                | lambda


# Producción que genera un movimiento
9. MOVIMIENTO -> movimiento
               | lambda


# Producción que genera un símbolo
10. SIMBOLO -> simbolo
             | lambda


# Producción que genera comentarios que van inmediatamente después del segundo movimiento.
11. COMENTARIO -> lllave CONTENIDO rllave
                | lparen CONTENIDO rparen
                | lambda
```

```
# Producción que genera el contenido interno de los comentarios.
# Un comentario puede tener cualquier cadena en su interior, con o sin comentarios anidados.
12. CONTENIDO -> CONTENIDO CADENA CONTENIDO
                | COMENTARIO
                | lambda


# Producción para generar las cadenas que están adentro de los comentarios. Puede ser un texto o
13. CADENA -> TEXTO CADENA
            | JUGADA_EN_COMENTARIO CADENA
            | lambda


# Producción que genera un texto.
14. TEXTO -> texto


# Producción que genera un movimiento adentro de un comentario.
15. JUGADA_EN_COMENTARIO -> movimiento


# Producción que genera el resultado de una partida de ajedrez
16. RESULTADO -> res
               | lambda
```

3.1. Dificultades encontradas y decisiones tomadas.

Al escribir las anteriores producciones se tomaron las siguientes decisiones:

- El archivo PGN no podía ser vacío. Tenía que haber, al menos, una partida.
- Cada partida debía tener, al menos, una jugada.
- Tenía que haber, al menos, un descriptor de evento.

La principal dificultad que tuvimos a la hora de proponer las producciones pertenecientes al parser fue al momento de calcular el máximo anidamiento de comentarios sin captura. En iteraciones anteriores a la solución final, el lexer no podía distinguir movimientos en comentarios de jugadas, lo cual nos llevó a realizar la solución a dicho caso explicada en la Sección 2. Una vez realizado este cambio, el único obstáculo restante fue poder diferenciar los movimientos que son capturas de los que no lo son, por lo que decidimos dividir la **CADENA** dentro de **CONTENIDO** la cual pueda ser tanto **TEXTO** como **JUGADA_EN_COMENTARIO**. Finalmente, la diferenciación de ambos se realiza mediante atributos, consultando cuál movimiento tiene o no una 'x', y, en caso afirmativo, tratarlo como captura. A continuación, se explicará más en detalle el uso de los atributos.

3.2. Manejo de errores

Cuando el parser identifica un error se levanta una excepción para detener la evaluación de la cadena. Luego, el error es atrapado y, simplemente, se lo muestra por salida estándar junto con información contextual del mismo (numero de linea, token, etc).

3.3. Atributos

Los atributos son la parte fundamental del análisis de la semántica de una gramática. En nuestra solución utilizamos todos atributos **sintetizados**, los cuales tienen como principal rol controlar que se respete la estructura de los archivos PGN como, por ejemplo, la numeración de las jugadas y el máximo valor de anidamiento de un comentario que no contiene una captura en su cadena.

Para el caso del requerimiento especifico del máximo nivel de anidamiento se utilizó un atributo para identificar si el contenido de un comentario tenía capturas o no, y un atributo para contar el nivel de anidamiento. Los tokens del tipo **TEXTO** no contienen capturas y los movimientos solo la tenían si contenían una ”x”. El nivel de anidamiento se incrementaba en una unidad cuando ocurría alguno de los siguientes casos:

- El comentario no tenía capturas.
- El comentario tenía capturas pero tenia en su interior un comentario que no las contenía. Por lo tanto, al ser independientes las capturas, el nivel del mas interno debía ser incrementado.

En los casos en donde había varias producciones posibles de comentarios del lado derecho se tomaba siempre el máximo y este valor era siempre ”ascendido” en el árbol hasta la producción inicial, donde se imprimía el valor máximo. Para el resto de las verificaciones semánticas, como la numeración y la continuación de las jugadas, se usaron atributos sintetizados que contenían los valores numéricos y se comparaba en los casos pertinentes los valores de los mismos. Además, se asumió que solo la última jugada de la partida podía tener un solo movimiento.

4. Ejemplos

Los siguientes ejemplos corresponden a corridas del programa desarrollado con diferentes cadenas de texto, mostrando errores en el análisis léxico de la cadena, errores en el análisis sintáctico y semántico, y también casos de cadenas reconocidas como válidas de PGN:

- **Casos válidos**:

```
>python3 parser.py
[Evento "asdasdasd"]

1. e6 e6 2. exa5 e8 1-0
Máximo nivel sin capturas: 0.
```

```
>python3 parser.py
[Evento "a"]
[Evento "b"]

1. a4 {esto es un comentario (exa6 capturado)} 1... P4xh3 1/2-1/2
Máximo nivel sin capturas: 1.
```

```
>python3 parser.py
[asd "b"]
[descriptor "c"]

1. a4 a3 2. a1 h6 0-1

[slpe "mabo"]
[rpa "cc"]
[vb "ccc"]

1. a3 {ss{ssss(exa6{sin capturas})}} 1... a8 2. e3 b3 3. a4 a6 1-0
Máximo nivel sin capturas: 4.
```

- **Error del lexer**: Los errores del lexer se producen en el momento del análisis léxico de la entrada. En este ejemplo se puede observar que el movimiento final 'Qd6Q' no cumple con el token definido como posible **MOVIMIENTO**, visto en la Sección 2. Al no poder matchear con ninguna definición de los tokens, el lexer levanta la excepción default por error.

```
>python3 parser.py
[Elemento "x"]

1. e4 { Muevo primero a e6 (luego jacke (mate)) } 1... e6 2. Bd3 Qg7
3. Qd6Q 1-0
Carácter inválido 'Q' en linea 4
```

  En el siguiente caso la entrada no cumple con la definición del token **DESCRIPTOR**, el cual debe empezar con cualquier cadena de texto sin comillas, seguido por cualquier cadena entre comillas.

```
>python3 parser.py
["a" b]

1. e4 d5 1/2-1/2
Carácter inválido '[' en linea 1
```

- **Errores sintácticos**: Los errores sintácticos se producen cuando la cadena no puede ser generada a partir de la gramática. En este caso se ve que la jugada 2 tiene 3 movimientos, lo que es un error.

```
>python3 parser.py
[Partida "uno"]

1. Qg4 xf5 {Movimiento intenta {comer pa n f5}} 2. Nxe2 Rg8 a2
3. e1 b5 0-1
Error de sintaxis: 'a2' en la linea 3
```

- **Errores semánticos**: Los siguientes casos especifican diferentes excepciones capturadas por el parser

```
>python3 parser.py
[a "b"]

1. e4 d5 3. Rf8 gg7 1-0
Error. Los numeros de jugada no son secuenciales: 3 y 1 en linea 3
```

```
>python3 parser.py
[a "b"]

1. e4 d5 2. Bxd1 {hola} 3... Pa6 0-1
Error. Continuacion de jugada incorrecta: 3 y 2 en linea 3
```

```
>python3 parser.py
[iii "iqisi"]
[akl "klakls"]

1. e4 d5 2. f4 Nxf7 3. d7 4. fb7 b7 0-1
Error. Solo el ultimo movimiento puede jugar solo negras o blancas:
En jugada 3 en linea 4
```

5. Conclusión

En el presente trabajo pudimos desarrollar la totalidad del análisis léxico, sintáctico y semántico de los archivos PGN, devolviendo el máximo nivel de anidamiento de comentarios sin capturas. Pudimos observar cómo llevar a la práctica los conocimientos vistos en clase sobre estas primeras etapas que realiza un compilador. Se pudo comprender la utilidad del formalismo de las gramáticas libres de contexto, en especial de las LALR, y de las gramáticas de atributos para poder realizar estos tipos de análisis, ya que sin ellas hubiera sido más difícil.

En cuanto a su complejidad, sabemos que el parsing LALR es O(n). Además, basado en la documentación, no incorporamos muchas reglas de gramática dentro de una misma función del parser, para que no existan varias estructuras de control condicionales, ya que el parser "sabe" en que regla entró. Por lo tanto, se prefirió, por cuestiones de performance, usar la mayor cantidad posible de funciones sin sacrificar claridad.