DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico

Lexer y Parser

Teoría de Lenguajes
Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Alexis Wolfsdorf	529/17	alexiswolfsdorf@gmail.com
Alvaro Machicado	005/11	rednaxela.007@hotmail.com.ar
Julián Recalde Campos	502/17	recaldej@hotmail.com.ar

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

1. Introducción

Un archivo PGN es una colección de partidas de ajedrez con datos sobre las mismas, como los jugadores, eventos involucrados y comentarios a lo largo de las jugadas.

El objetivo de este trabajo es poder crear un parser que nos permita reconocer archivos PGN, validarlos e imprimir información relevante sobre lo leído, como por ejemplo, la cantidad de piezas que se fueron perdiendo (*capturas*).

Si bien se pedía trabajar sobre partidas simplificadas, una vez que alcanzamos ese punto, no nos pareció muy complicado extender lo que teníamos para permitir la lectura de partidas reales como las que se pueden descargar de `pgnmentor.com`.

Estamos bastante satisfechos con lo que llegamos a hacer.

2. Descripción Solución

Para afrontar las dificultades que trae el parseo de lenguaje PGN decidimos usar las herramientas provistas por PLY que trae consigo su propio lexer (LEX) y analizador sintáctico (YACC) escritos en python y que usamos para armar un parser basado en LR el cual es rápido, eficiente en memoria y bueno para gramáticas grandes. Actualmente, PLY construye tablas usando el algoritmo de LALR(1).

Adjuntamos el código en la entrega que debería contener:

- `main.py`: Todo el trabajo que hicimos. Aquí importamos el lexer y yacc de PLY para armar los tokens y producciones asociadas.
- `lex.py`: Lexer de PLY
- `yacc.py`: Yacc de PLY
- Carpeta con ejemplos que son archivos PGN
- `requirements.txt`: Archivo necesario para importar las dependencias de nuestro proyecto (solo tenemos la dependencia `click`).

2.1. ¿Cómo ejecuto el proyecto?

En la sección **Ejemplos de Corrida** se pueden ver algunos ejemplos de los comandos que ejecutamos pero antes necesitamos hacer una configuraciones para cumplir con los requisitos mínimos y así poder ejecutar el proyecto:

1. Se debe contar con al menos la versión de Python 3.7 instalada para que funcione (no probamos que funcione con versiones anteriores).
2. Parados en el directorio del proyecto ejecutar `pip install -r requirements.txt`
3. `cd pgn-validator/`

Con esto ya deberían estar listos para probar el trabajo que hicimos.
Ver sección **Ejemplos de Corrida** para ver los comandos disponibles.

3. Gramática

3.1. No Terminales

Símbolo	Descripción
P	Impresión de resultados
F	Archivo PGN
H	Encabezado
G	Partida
M	Movida de 2 jugadores en conjunto con el número de jugada
C	Comentario
C1	Contenido de comentario

3.2. Terminales

Símbolo	Descripción
h	Item encabezado
m	Movida de un solo jugador
r	Resultado
n	Número jugada
nc	Continuación de número jugada
t	Texto
{	Llave abriendo
}	Llave cerrando
(	Paréntesis abriendo
)	Paréntesis cerrando

3.3. Atributos

Nombre	Tipo	Descripción
F.list_cant_capt	Sintetizado	Listado de Cantidad de capturas
G.cant_capt	Sintetizado	Cantidad de capturas
M.cant_capt	Sintetizado	Cantidad de capturas
C.cant_capt	Sintetizado	Cantidad de capturas
C1.cant_capt	Sintetizado	Cantidad de capturas
M.move_no	Sintetizado	Número de jugada
m.value	Sintetizado	Número de m
m.is_capture	Sintetizado	True si es una captura, False si no

3.4. Producciones

$P \longrightarrow F$	{ print(reverse(F.list_cant_capt)) }
$F1 \longrightarrow H\ GF2$	{ F1.list_cant_capt = append(F2.list_cant_capt, G.cant_capt) }
$F \longrightarrow H\ G$	{ F.list_cant_capt = list(G.cant_capt) }
$G \longrightarrow M\ r$	{ G.cant_capt = M.cant_capt, COND(M.move_no == 1) }
$H1 \longrightarrow h\ H2$	{ }
$H \longrightarrow h$	{ }
$M1 \longrightarrow n\ m1\ m2\ M2$	{ M1.cant_capt = int(m1.is_capture) + int(m2.is_capture) + M2.cant_capt, M1.move_no = n.value, COND(n.value == M2.move_no - 1) }
$M1 \longrightarrow n\ m1\ m2\ C\ M2$	{ M1.cant_capt = int(m1.is_capture) + int(m2.is_capture) + C.cant_capt + M2.cant_capt, M1.move_no = n.value, COND(n.value == M2.move_no - 1) }
$M1 \longrightarrow n\ m1\ C\ nc\ m2\ M2$	{ M1.cant_capt = int(m1.is_capture) + C.cant_capt + int(m2.is_capt) + M2.cant_capt, M2.move_no = n.value, COND(n.value == nc.value), COND(n.value == M2.move_no - 1) }
$M1 \longrightarrow n\ m1\ C1\ nc\ m2\ C2\ M2$	{ M1.cant_capt = int(m1.is_capture) + C1.cant_capt + int(m2.is_capture) + C2.cant_capt + M2.cant_capt, M1.move_no = n.value, COND(n.value == m2.move_no - 1), COND(n.value == nc.value) }
$M \longrightarrow n\ m$	{ M.cant_capt = int(m.is_capture), M.move_no = n.value }
$M \longrightarrow n\ m\ C$	{ M.cant_capt = int(m.is_capture) + C.cant_capt, M.move_no = n.value }
$M \longrightarrow n\ m1\ m2$	{ M.cant_capt = int(m1.is_capture) + int(m2.is_capture), M.move_no = n.value }
$M \longrightarrow n\ m1\ C\ nc\ m2$	{ M.cant_capt = int(m1.is_capture) + C.cant_capt + int(m2.is_capture), M.move_no = n.value, COND(n.value == nc.value) }
$M \longrightarrow n\ m1\ m2\ C$	{ M.cant_capt = int(m1.is_capture) + int(m2.is_capture) + C.cant_capt, M.move_no = n.value }
$M \longrightarrow n\ m1\ C1\ nc\ m2\ C2$	{ M.cant_capt = int(m1.is_capture) + C1.cant_capt + int(m2.is_capture) + C2.cant_capt, M.move_no = n.value, COND(n.value == nc.value) }
$C \longrightarrow \{CB\}$	{ C.cant_capt = CB.cant_capt }
$C \longrightarrow (CB)$	{ C.cant_capt = CB.cant_capt }
$CB1 \longrightarrow t\ CB2 \mid n\ CB2 \mid nc\ CB2 \mid r\ CB2$	{ CB1.cant_capt = CB2.cant_capt }
$CB1 \longrightarrow m\ CB2$	{ CB1.cant_capt = int(m.is_capture) + CB2.cant_capt }
$CB1 \longrightarrow C\ CB2$	{ CB1.cant_capt = CB2.cant_capt }
$CB \longrightarrow t \mid n \mid nc \mid r$	{ CB.cant_capt = 0 }
$CB \longrightarrow m$	{ CB.cant_capt = int(m.is_capture) }
$CB \longrightarrow C$	{ CB.cant_capt = C.cant_capt }

4. Decisiones y problematicas

En esta sección nos dedicaremos explayar todos los problemas a los que nos fuimos enfrentando, como los resolvimos y decisiones que tomamos al momento de determinar si una partida es válida o no.

4.1. Tokens en comentarios

Para resolver algunos problemas con la interpretación de tokens dentro de los comentarios, decidimos que el *lexer* analice cada palabra en el comentario intentando asociarla a los tokens que definimos. Si no hay token que se pueda asociar a la palabra entonces lo más probable es que se termine asociando al token texto: el token texto acepta todos los caracteres que no son *whitespace* o caracteres resevados (llaves y parentesis) como parte de una palabra.

Este método nos permite identificar, dentro de los comentarios, las movidas, y en ellas, aquellas que representan capturas de piezas que son las que nos interesan considerar en la **suma total de capturas de una partida**.

Cualquier otro tipo de token reconocido, no nos interesa.

4.2. Tokens prefijos de otros Tokens: h1 vs h1=Q

Debido a que las movidas que pueden hacer los jugadores pueden representarse de formas muy variadas, la expresión regular que las *tokeniza*, es bastante compleja.

Lo que nos terminó pasando es que se empezaron a producir **problemas de prefijos** como por ejemplo: una movida de coronación teniendo como prefijo una movida que representa el movimiento de un peón. Estos casos se *tokenizaban* mal y producían conflictos dado que ambos son token válidos.

Para resolver este problema le exigimos al *lexer* que *tokenice* primero las palabras mas complejas (largas) y luego las mas cortas reorganizando la expresión regular.

Otro ejemplo de esto son las enumeraciones que podían aparecer con el formato 2. o 2... Si bien este caso ya no fue a nivel expresión regular (sino a nivel token), la solución fue la misma: darle prioridad al token más complejo sabiendo que el *lexer* prioriza los tokens en el orden en el que están escritos en el código.

4.3. No poder usar variables heredadas

Ya al momento de querer imprimir los resultados para aquellos casos en los que un archivo PGN contiene varias partidas, notamos que los resultados se imprimían en el orden inverso al que aparecían en el archivo.

Esto se debe a por como está implementado *yacc*: en las producciones se resuelven primer los no terminales hijos antes que el terminal padre por lo que no se les puede pasar ninguna variable (heredar). Son gramáticas forzosamente **S-atribuidas**.

En nuestro caso, dada esta propiedad y que se trata de una producción con recursión a derecha (los hijos terminaban siendo las últimas partidas del archivo) se produce el problema descrito al principio.

Para 'emular' una variable heredada, lo que hicimos fue listar todos los resultados y revertirlos antes de imprimirlos.

En la gramática de atributos nos tomamos la libertad de abusar de la notación para representar mejor lo que hicimos.

Definimos una lista usando la notación `list(some_value)` que crea una lista con `some_value` como primer elemento.

También definimos una función `append(list, some_other_value)` que agrega `some_other_value` al final de la lista `list` y una función `revert(list)` que invierte la lista `list`.

4.4. Conflicto de definiciones

Debido a que lo que nos indicaron en la presentación del TP conflictuaba con los casos válidos/inválidos que nos pasaron en los tests, tomamos algunas decisiones.

4.4.1. Partidas sin *Header*

En caso de que una partida no tenga ningún *header*, esa partida sera invalida. Si nos pasan un archivo con 10 partidas, las 10 deben tener su *header* justo antes del listado de movidas.

4.4.2. *Headers* contiguos

Ignoramos los *newlines* y los espacios si están por fuera de un token. Es por esto que si tenemos 2 *headers* seguidos sin *newline* los consideraremos como válidos.

5. Ejemplos de corridas

Probamos varios ejemplos mientras armábamos el TP y es por eso que implementamos una mecánica que nos permite indicar si vamos a leer un archivo o un partida de ejemplo por parámetro (*inline*).

Algunos de los ejemplos se encuentran en la carpeta *examples* del código entregado.

5.1. Ejemplo 1: Ejemplo *inline*

Tras ejecutar

```
python3 ./main.py process-str '[Header "Header content"] 1.exd4 0-1'
```

el output es:

```
Procesando ...
cantidad de capturas: 1
```

5.2. Ejemplo 2: Ejemplo enunciado

Tras ejecutar

```
python3 ./main.py process-file examples/ejemplo\_enunciado.pgn
```

Se levantará el archivo que contiene el ejemplo que nos pasaron en el enunciado del TP con el output:

```
Procesando ...
cantidad de capturas: 14
```

5.3. Ejemplo 3: MacKenzie

Descargamos uno de los archivos de la pagina de la bibliografía y tras ejecutarlo:

```
python3 ./main.py process-file examples/MacKenzie.pgn
```

con el output:

```
Procesando ...
Partida 1. Cantidad de capturas: 14
Partida 2. Cantidad de capturas: 6
Partida 3. Cantidad de capturas: 8
Partida 4. Cantidad de capturas: 17
Partida 5. Cantidad de capturas: 16
Partida 6. Cantidad de capturas: 12
Partida 7. Cantidad de capturas: 23
...
Partida 190. Cantidad de capturas: 19
Partida 191. Cantidad de capturas: 14
Partida 192. Cantidad de capturas: 15
Partida 193. Cantidad de capturas: 16
Partida 194. Cantidad de capturas: 8
Partida 195. Cantidad de capturas: 20
Partida 196. Cantidad de capturas: 24
Partida 197. Cantidad de capturas: 15
Partida 198. Cantidad de capturas: 21
```

6. Conclusiones

Tras haber realizado un parser+lexer que se encargue de procesar cadenas que representan partidas de jugadas, podemos concluir que llevar a cabo la tarea de reconocer lenguajes tiene sus complejidades y sus trucos. la cadena se debe poder interpretar de una única forma, por lo tanto las gramáticas deben no ser ambiguas. Por otro lado hay que priorizar siempre que tokens se traducen antes que otros y tener bien en claro que exista un orden posible para el problema que queremos resolver, por ejemplo, si en un contexto quisiéramos que se lea un token pero que tenga otro comportamiento en otro contexto sería problemático y deberíamos arreglar la gramática y/o el lexer. Finalmente reconocer lenguajes es una tarea compleja que requiere de prestar atención al los detalles de cada lenguaje que se quiera reconocer. Luego de superar estas complicaciones, uno puede logra entender un lenguaje no ambiguo sintáctica y semánticamente para luego hacer que una computadora pueda hacerlo también.

'cualquiera puede parsear'
Pero me doy cuenta... recién ahora comprendo sus palabras: no cualquiera puede convertirse en un gran parser, pero un gran parser sí puede provenir de cualquier lugar.
- Julián Recalde Campos -