Autómatas, Teoria de Lenguajes y Compiladores

Trabajo Práctico Especial 2

Analizador de Partidas de Ajedrez

Castiglione, Gonzalo	49138
Susnisky, Darío	50592
Ordano, Esteban	50753
Sturla, Martín	50684

26 de noviembre de 2011

Índice

1. Resumen

El trabajo práctico consistió en generar un analizador sintáctico de partidas de ajedrez en formato *PGN* para luego mostrar un tablero y poder hacer un seguimiento del partido.

2. Consideraciones realizadas

Notese que para ejecutar el programa *chess* es necesario indicar por entreada estándar cuál es el archivo a leerse. Si se quisiese ejecutar el archivo "prueba.pgn" que se encuentra en la misma carpeta que *chess* habría que ejecutar "chess <prueba.pgn".

Notese que las imagenes para que la representación gráfica se vea de manera correcta se encuentran en /res/images y esto es un requerimiento para el correcto funcionamiento.

2.1. Formato PGN

Como indica el enunciado, las partidas de ajedrez DEBEN tener el formato *STG*. Además no se aceptan espacios de más, comentarios o anotaciones de variantes recursivas. Las etiquetas DEBEN aparecer en el orden Event, Site, Date, Round, White, Black, Result. Entre las etiquetas y las jugadas en formato *SAN* debe haber un solo fin de línea y nada más (Nota: Se aceptan los finales de línea tanto de Windows como de Linux).

2.2. Validez de jugadas

Existe una cierta validación de jugadas por parte del analizador, dado que se debe encontrar la pieza que se está moviendo. Esto implica que no pueden existir movimientos imposibles en cuanto a la naturaleza del movimiento de las piezas. Por otra parte, validaciones más complejas como por ejemplo restricciones para enrocar no son validadas.

2.3. Interfaz gráfica

Se optó por usar una interfaz gráfica que requiere librería SDL. Por lo tanto para compilar se deben instalar dos paquetes de la librería SDL; la librería estándar y la librería *image*. En particular, en Ubuntu son las librerías *libsdl1.2-dev* y *libsdl-image1.2-dev*. Debido a estas dependencias que se agregaron, se decidió incluir el archivo compilado en la entrega final.

3. Descripción del desarrollo

Al igual que con el trabajo especial previo, al comenzar tratamos de diferenciar los distintos módulos de trabajo.

Una parte importante del trabajo implicaban leer el archivo .PGN, y validarlo con un *parser* adecuado. Luego, era necesario utilizar un analizador sintáctico para obtener los datos en distintos *tokens*, ubicarlos en distintas estructuras que fueron definidas con el proposito de poder procesar las jugadas según las reglas del ajedrez. Finalmente, un módulo se encargaba del *frontend* que implica la presentacion gráfica de la aplicación.

3.1. Gramatica usada

Notas: Los símbolos terminales son las palabars en mayúscula, que son *tokens* devueltos por *lex*. Los no terminales son aquellas en minúscula.

program ⇒ option program | game

option ⇒ INITIAL_TOKEN STRING END_TOKEN

option ⇒ INITIAL_DATE_TOKEN INTEGER '.' INTEGER '.' INTEGER END_TOKEN

option ⇒ INITIAL_ROUND_TOKEN INTEGER END_TOKEN

game ⇒ round SPACE move SPACE move SPACE game | round SPACE move SPACE FINALRESULT | FINALRESULT | λ

round ⇒ ROUND

move ⇒ castle check | normal_move check pawn_move check

normal_move ⇒ PIECE COL ROW | PIECE CAPTURE COL ROW | PIECE COL COL ROW

normal_move ⇒ PIECE COL CAPTURE COL ROW | PIECE ROW COL ROW | PIECE ROW CAPTURE COL ROW

pawn_move ⇒ COL ROW | COL ROW COL ROW | COL CAPTURE COL ROW | COL ROW CAPTURE COL ROW

pawn_move ⇒ COL ROW CROWN PIECE | COL CAPTURE COL ROW CROWN PIECE

pawn_move ⇒ COL ROW CAPTURE COL ROW CROWN PIECE | COL ROW COL ROW CROWN PIECE

castle ⇒ SHORTCASTLE | LONGCASTLE

check ⇒ CHECK | CHECKMATE | λ

3.2. Atributos de símbolos no terminales

La mayoría de los *tokens* devueltos por *lex* tienen un valor asociado reperesntando qué es lo que se leyó. Por ejemplo, el atributo COL tiene el valor de la columna, ROW el valor de la fila, PIECE el carácter representando la pieza, etc. De los símbolos no terminales, los únicos con valor son normal_move, pawn_move, move, castle, check y round. Round posee el valor númerico de la ronda, check un valor representando si hubo jaque, jaquemate o nada, y castle un valor representando si el enroque fue corto o largo. Normal_move, pawn_move y move poseen una estructura de tipo movimiento que posee muchos atributos, entre ellos, la pieza, su destino, si hubo jaque o no, enroque, etc. En las reescrituras del símbolo game, se llenan en esats estructuras devueltas por los movimientos asignándoles el color y se guardan en un vector para luego ser usadas por el *front-end*.

3.3. Lógica interna de la aplicación

Al comenzar a armar las estructuras a usarse en la lógica interna de la aplicación, contamos como basico contar con estructuras para representar el tablero y estructuras para representar las fichas. Después de cierto debate, decidimos apropiado que nuestro tablero sea simplemente un vector de las 32 fichas, agregando dentro de la estructura ficha variables representando la fila y la columna. Esto se decidió ya que los casilleros no tienen ninguna propiedad en particular más que contener fichas o su color (sencillamente calculable).

Las fichas, contienen (además de su fila y su columna actual) su identificador único, su tipo de ficha, su color y si se encuentra viva.

Acompañadas de estas estructuras contabamos con metodos sencillos de representar los colores y los tipos de ficha (*enums*).

Por otra parte, contamos con ciertas estructuras que representan lo que se lee del archivo de origen, estas son las estructuras que representan cada movimiento y la que representa el encabezado de cada archivo. Estas estructuras se definieron de forma sencilla a partir de los *tokens* dados por el analizador sintactico.

3.4. Lógica de la aplicación

La lógica de la aplicación estaba dada por las reglas del ajedrez y simplemente consistía en corroborar que los movimientos dados por el archivo respetasen estas reglas y ejecutar este movimiento para que tanto nuestras estructuras internas esten actualizadas según estos movimientos.

A continuación se presentan algunas cuestiones interesantes que debían ser validadas en este punto de la aplicación:

- Detectar la ficha que se estaba moviendo.
- Que el movimiento sea valido para el tipo de pieza.
- Capturas.
- Coronaciones.
- Enroques.

4. Dificultades encontradas

Al crear gramáticas con *bison* es posible encontrarse con conflictos de tipo *shift-reduce*, que es justamente cuando una entrada en la tabla LALR(1) se puede tomar la acción de reducir o ir a otro estado (goto). En estos casos, *bison* no compila. Se puede usar la etiqueta *expect n* donde n es la cantidad de conflictos presentes, para que compile. Sin embargo, lo único que hará *bison* es tomar la acción *default* de ir a otro estado, es decir, no reducir. Esto no es siempre lo que se quiere. Idealmente uno debe evitar este tipo de conflictos. Se identificaron tres soluciones para resolver este tipo de conflictos:

- Cambiar la gramática, cambiando el lexer y agregando símbolos que toman varios de los símbolos anteriores. Esto sería darle más trabajo al lexer. También se puede hacer uso de los estados que usa el lexer, ya sea los estados izquierda (aquellos encerrados por $<$ o $>$) o estados de tipo derecha (pedir, por ejemplo, la cadena $abd|eff$, que signfica pedir la cadena abd si y solo si está seguida por la cadena eff).
- Cambiar el parser (*yacc*) y hacer reglas eliminando recursiones o reglas dispuestas distintas para que no conflictúen. Esto sería darle más trabajo al parser.
- Manejar el problema en la parte semántica, es decir, resolverlo en el código C. Esto es darle más trabajo al lenguaje que se está usando.

En el desarrollo del trabajo se intentó dividir el trabajo entre estas tres herramientas en una manera que el trabajo de los tres no sea ni trivial ni extremadamente complicado, priorizando la prolijidad del código. Finalmente se logró que la gramática resultante no tenga conflictos.

5. Futuras Extensiones

5.1. Consideraciones

Como ya se ha explicado en una sección anterior, se asumen muchas cosas en cuanto al formato de entrada del archivo. Se podría modificar la gramática para ser más laxos en cuanto a qué se acepta y qué no. Por ejemplo, se podrían aceptar las opciones en cualquier orden, permitir espacios de más, etc. Desde ya esto agregaría una complejidad apreciable a la gramática usada y a las reglas de sus producciones.

5.2. Formato

Se podrían aceptar comentarios o más opciones que las siete requeridas. Además, sería interesante poder mostrar comentarios en sus jugadas respectivas.

5.3. Retroceso

El programa está diseñado para ir solo adelante con las jugadas. Se podría implementar que con una cierta tecla se vaya hacia adelante y con otra hacia atrás, de esa manera se podría hacer un seguimiento del partido con mayor libertad.

5.4. Marcado de movimientos

Se podría marcar el casillero destino y fuente de la última pieza movida de tal manera de simplificar el seguimiento del partido.