MODELLALAPÚ SZOFTVERFEJLESZTÉS – 3. GYAKORLAT – ANTLR SEGÉDLET

Szöveges szakterületi nyelvek a gyakorlatban

Somogyi Ferenc Attila

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

1117 Budapest, Magyar tudósok körútja 2.
Q. ép. B. szárny 207. • www.aut.bme.hu
Telefon: 463-2870 • Fax: 463-2871

I. Bevezetés

A gyakorlat célja, hogy a gyakorlatban is megismerkedjünk a klasszikus fordítás első három fázisával. Ehhez az ANTLR parser generátort fogjuk használni. A gyakorlat során megismerkedünk az ANTLR alapszintű használatával:

- ANTLR használata IntelliJ IDEA környezetben
- Nyelvtan definiálása: lexer és parser szabályok
- Levezetési fa megjelenítése tetszőleges bemenet alapján
- Egyszerű szemantikai elemző elkészítése (szimbólumtábla, szemantikai hibák)
- Sakk variánsokat leíró nyelv készítése a fentiek alapján

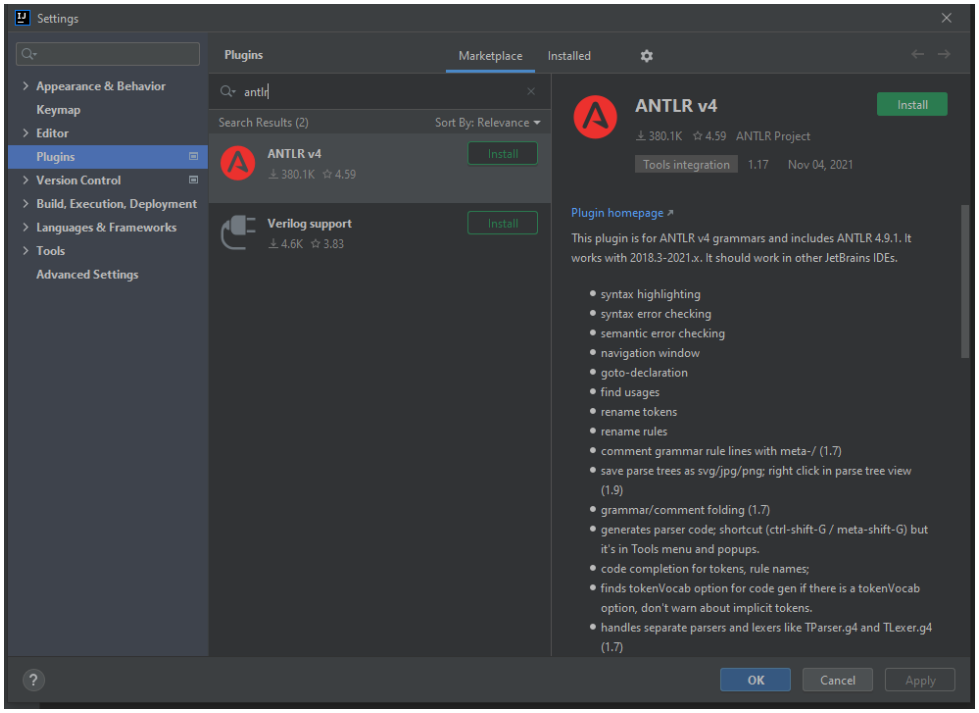
Fontos: jelen útmutató (valamint a későbbiek is) az **ANTLR 4**-es verziójához készült. Az **ANTLR 3**-as verziója sok szempontból hasonló a 4-eshez, viszont sok szempontból nem. Az interneten fellelhető segédanyagoknál és kérdéseknél (pl. stackoverflow) érdemes meggyőződni róla, hogy melyik verzióról van szó.

ANTLR beüzemelése IntelliJ környezetben

Töltsük le az **IntelliJ IDEA Community Edition**-t: https://www.jetbrains.com/idea/download/#section=windows

JDK-ra is szükségünk lesz, amit például innen tölthetünk le (alternatívaként az IntelliJ új Java projekt létrehozásakor automatikusan felajánlhat más JDK-t is, az is jó): http://www.oracle.com/technetwork/java/javase/downloads/index.html

Erősen ajánlott feltelepítenünk az IntelliJ ANTLR plugint – különben pl. a lexer és parser generálást kézzel kell csinálnunk. A plugin segítségével az IntelliJ számos hasznos funkciót ad ANTLR nyelvtanok szerkesztéséhez és „debugolásához”. A plugint egyszer kell feltelepítenünk, utána minden projektnél elérhető. ***File --> Settings...*** menüpont, válasszuk oldalt a ***Plugins***-t, keressünk rá az ANTLR-re, és telepítsük a plugint (utána indítsuk újra az IntelliJ-t):



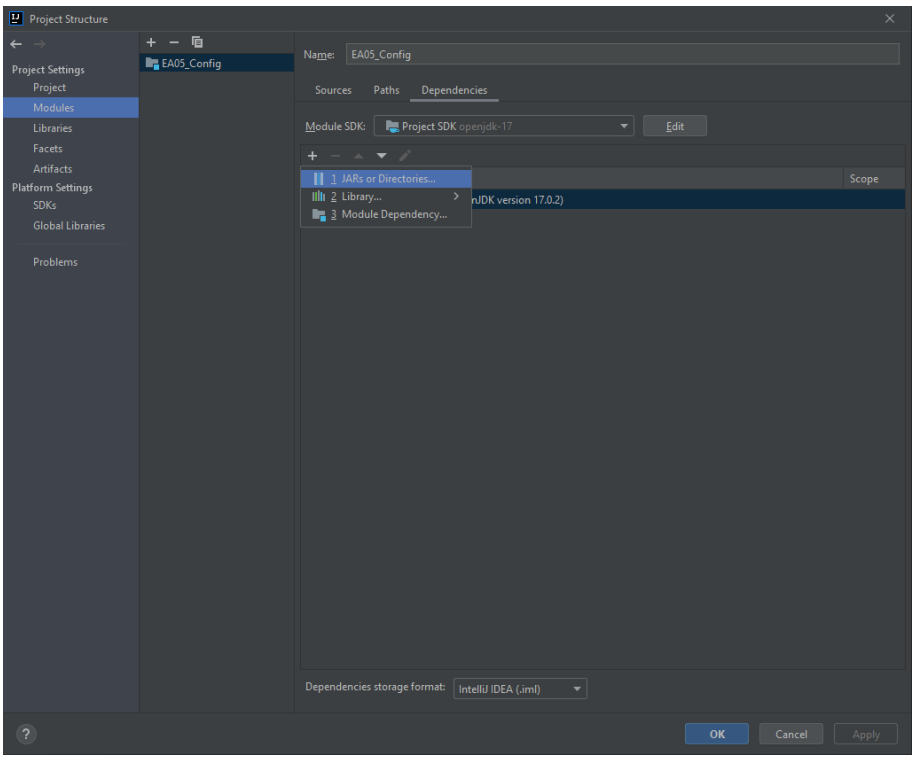
Az ANTLR többféle célnyelvet támogat (pl. C++, C#, Java, Python, JavaScript), illetve több fejlesztőkörnyezethez is létezik hozzá integráció (pl. Visual Studio Code). Mi Java-val és IntelliJ-vel fogunk dolgozni, de választhatnánk mást is.

Ezután nyissuk meg a kiinduló projektet (**GY03_ChessVariants**)! A **libs** mappa alatt megtalálható az ANTLR runtime (egy *jar* fájl), ami a lexer és parser generáláshoz, illetve a runtime funkciók eléréséhez (pl. szintaxisfa felépítése runtime) szükséges. A *chess* package-ben két fájl található:

- **chess_input.txt**: egy példa bemeneti szöveget tartalmaz; ezt fogjuk később feldolgozni
- **ChessVariantsRunner.java**: egy *main* függvényt tartalmaz; itt fogjuk később használni a runtime jar-t és feldolgozni a bemeneti szöveget a generált lexer és parser segítségével
- **Symbol.java** és **SymbolTable.java**: általános szimbólumtábla nagyon alapszintű megvalósítása (pl. scoping és típusrendszer nincs benne)

Amennyiben új projektet hozunk létre, akkor a runtime *jar* fájlt kézzel (vagy egy build rendszer segítségével) kell a projekthez csatolnunk a következő módon:

- Töltsük le az ANTLR runtime *jar*-t: https://www.antlr.org/download
- Másoljuk be a jar-t a projektünk fájlszerkezetébe (pl. **libs** mappába)
- Adjuk hozzá a projekthez függőségként a jar-t: **File --> Project Structure... --> Modules --> Dependencies**



A fenti lépéseket a kiinduló projekt használata esetén csak akkor kell megtenni, hogyha a Dependencies alatt nem ismerné fel magától a lib mappában található jar fájlt. Ezt mindenképpen ellenőrizzük!

II. ANTLR NYELVTAN

A **chess** package-n ***jobb klikk --> New --> File***, a nyelvtan neve legyen **ChessVariants.g4**. A g4 az ANTLR fájlformátuma, a „g" a grammar szóra utal, míg a „4" az ANTLR fő verziószámára. A következőkben először általánosságban nézzük meg egy ANTLR nyelvtan felépítését, majd megírjuk a sakk variánsokat leíró nyelv kiinduló nyelvtanát.

ANTLR NYELVTAN DEFINIÁLÁSA ÁLTALÁBAN

A nyelvtan elején mindig a nyelvtan neve áll a ***grammar*** kulcsszóval megadva, pontosvesszővel lezárva:

```
grammar ChessVariants;
```

Egy ANTLR nyelvtan szabályokból épül fel, hasonlóan a környezetfüggetlen (**CF**) nyelvtanokhoz. Az ANTLR szabályainak – a CF nyelvtanokhoz hasonlóan – bal és jobb oldala van. A szabályokat pontosvesszővel kell lezárni. A szabályok két csoportba oszthatók:

- **Lexer szabályok**: a jobb oldalon csak szöveg (*string*) állhat, aposztrófok között. A bal oldalon a szabály neve szerepel, amely a lexikai elemzés során a keletkező *token* neve is. A bal oldalt csupa nagybetűvel írjuk.
- **Parser szabályok**: a jobb oldalon *lexer*, illetve *parser* szabályok neve is állhat, tetszőleges kombinációban. A bal oldalon a szabály neve szerepel. A bal oldalt hagyományosan camelCase szerint írjuk.

A szabályok megadásánál a reguláris kifejezések világából ismerős operátorokat (**?**, *****, **+**, stb.) használhatjuk. Az ANTLR biztosít számunkra ún. parser akciókat (**parser action**) is. Ezek az akciók speciális utasításokat adnak a *parser* számára, amikor az a megadott szabályra illeszkedő szöveget talál. A *skip* akció segítségével például megmondhatjuk, hogy az adott szabály ne kerüljön bele a levezetési fába, azt a parser ugorja át.

Az ANTLR lehetőséget nyújt arra is, hogy a lexer és parser szabályokat külön nyelvtan fájlban adjuk meg. Ekkor a parserhez tartozó .g4 fájlban tudjuk importálni a lexerhez tartozó .g4 fájlt (ld. options és tokenVocab).

A CHESSVARIANTS NYELVTANÁNAK ELKÉSZÍTÉSE

A sakk variáns leíró nyelv a következő négy részből fog állni:

- Sakktábla méreteinek megadása
- Játékos körének leírása – opcionális
    - pl. első körben a játékos kettőt léphet
- Játékos figuráinak megadása
    - akár aszimmetrikus módon
- Figurák és szabályos lépéseik definíciója

A fentiek közül a klasszikus sakkban nem mind megengedett, a célunk új variánsok leírása. A nyelv kidolgozása során csak a fenti négy rész alap funkcionalitására törekszünk, természetesen nem egy éles nyelv lesz a végeredmény (a keretalkalmazást például meg sem írjuk hozzá). Jelen útmutató végén található néhány (opcionális) önálló feladat, amik irányt mutatnak, hogy hogyan lehetne továbbfejleszteni a nyelvet, miután az útmutató alapján elkészültünk.

Kezdjük néhány alapvető lexer szabály megírásával (ezek nagy részét – kicsit más formában – előadáson is láthattuk), amik több helyen is kellhetnek később:

```
INT: [0-9]+;
ID: [a-zA-Z_][0-9a-zA-Z_]*;
COMMENT: '//' ~[\r\n]* -> skip;
WS: [ \t\r\n] -> skip;
```

A lexer szabályok esetén fontos a sorrend, ugyanis az ANTLR által generált lexer sorrendben próbál meg tokeneket készíteni a lexémákból. Tehát az általánosabb szabályokat (mint a fentiek) tegyük a nyelvtan fájl végére.

A nyelv szerkezetének, illetve a sakktábla méreteinek megadására az alábbi parser szabályok szolgálnak:

```
chessVariant: boardDefinition turnDefinition? playerDefinition+ pieceDefinition+;
boardDefinition: BOARD dimensions;
dimensions: width X height;
width: INT;
height: INT;
```

A méret megadásához kapcsolódó lexer szabályok a következők:

```
BOARD: 'board';
X: 'x';
```

A játékos körét a következő parser szabályokkal írhatjuk le:

```
turnDefinition: TURN LPAREN turnStatement+ RPAREN;
turnStatement: TURN turnNumber COLON turnMoveStatement;
turnNumber: INT | X;
turnMoveStatement: MOVE numberOfPieces PIECE ;
numberOfPieces: (MAX)? INT;
```

Tehát a kör leírása utasításokból áll (*turnStatement*), melyek mindegyike leírja, hogy az N. körben (*turnNumber*) hány figurával (*numberOfPieces*) léphet a játékos. Ez később akár kiterjeszthető bonyolultabb logikával is. A kör leíráshoz kapcsolódó lexer szabályok (kivéve azokat, amiket korábban már megadtunk) a következők:

```
TURN: 'turn';
LPAREN: '(';
RPAREN: ')';
COLON: ':';
MOVE: 'move';
PIECE: 'piece';
MAX: 'max';
```

A játékosok megadása a következő parser szabályok alapján történik:

```
playerDefinition: PLAYER playerName (INVERT)? LPAREN piecePlacement+ RPAREN;
playerName: ID;
piecePlacement: pieceName coordinates;
coordinates: LBRACKET xCoord COMMA yCoord RBRACKET;
xCoord: INT;
yCoord: INT;
```

Az *xCoord* és *yCoord* szabályok akár el is hagyhatóak, de így a levezetési fa strukturáltabb lesz.

A játékosok megadásához kapcsolódó lexer szabályok (kivéve azokat, amiket korábban már megadtunk) a következők:

```
PLAYER: 'player';
INVERT: 'invert';
LBRACKET: '[';
RBRACKET: ']';
COMMA: ',';
```

Az 'invert' kulcsszó a figurák alapértelmezett kezdeti irányát fordítja meg; így adjuk meg, hogy a tábla tetején kezdő játékos figurái észak helyett déli irányba nézzenek. Ezt egy felokosított játékprogrammal akár el is kerülhetnénk.

A figurák és lépéseik leírására a következő parser szabályok szolgálnak:

```
pieceDefinition: PIECE pieceName (NOHIT)? pieceMoves;
pieceName: ID;
pieceMoves: LPAREN moveStatement+ RPAREN;
moveStatement: atomicStep (PLUS atomicStep)* (NOHIT)?;
atomicStep: direction moveAmount;
moveAmount: INT | ANY;
direction: NORTH | NORTHEAST | EAST | SOUTHEAST | SOUTH | SOUTHWEST | WEST | NORTHWEST | ANY;
```

A figurák leírásához kapcsolódó lexer szabályok (kivéve azokat, amiket korábban már megadtunk) a következők:

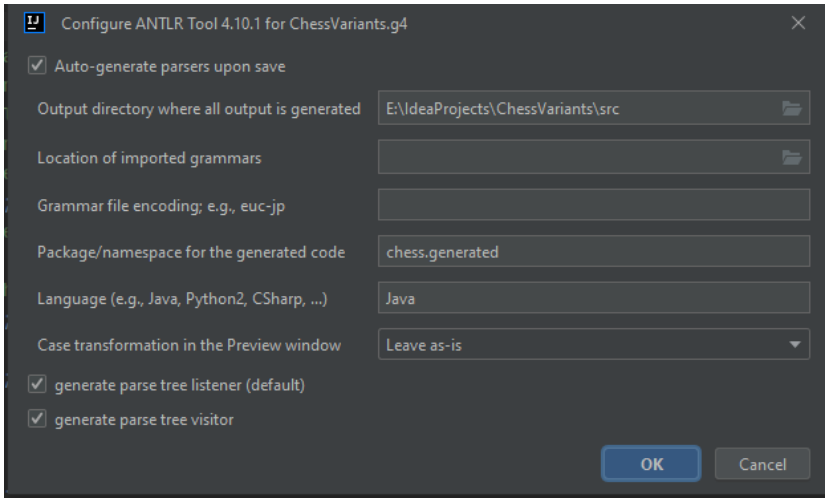
```
NOHIT: 'nohit';
PLUS: '+';
ANY: 'any';

NORTH: 'N';
NORTHEAST: 'NE';
EAST: 'E';
SOUTHEAST: 'SE';
SOUTH: 'S';
SOUTHWEST: 'SW';
WEST: 'W';
NORTHWEST: 'NW';
```

A 'nohit' kulcsszó figurák esetén azt jelenti, hogy nem lehet leütni (pl. király – ahol a sakk és matt kezelését a nehézsége nem írjuk le, ld. opcionális feladatok!); míg lépéseknél azt jelenti, hogy az adott lépés csak mozgás, leütni nem képes.

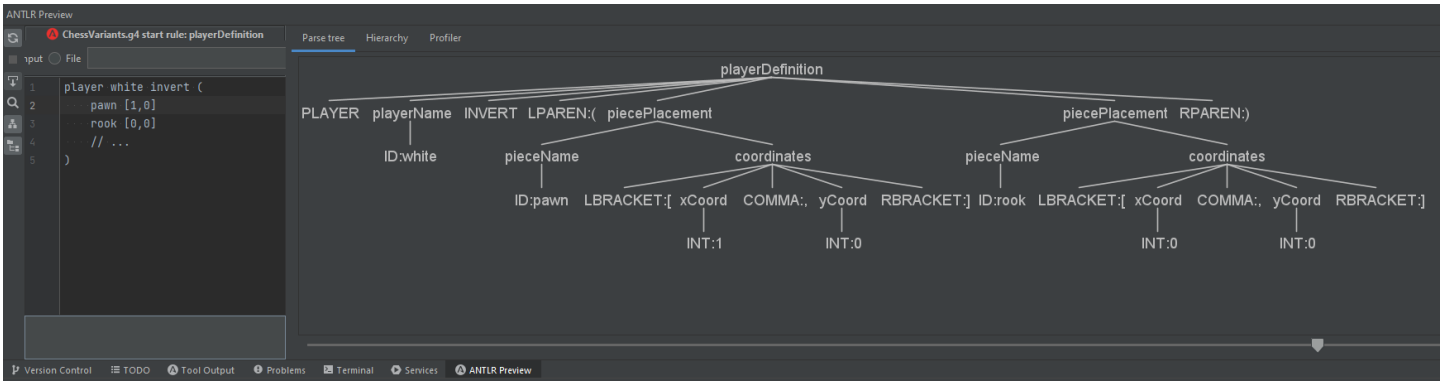
Lexer és parser generálása, nyelvtan debugolása

A nyelvtanból az ANTLR képes *lexert* és *parsert* generálni, amik automatikusan elvégzik helyettünk a lexikai, illetve a szintaktikai elemzést. Ez az ANTLR legfőbb funkciója. IntelliJ környezetben, ha a nyelvtan fájlban bárhol jobb klikkelünk, a ***Configure ANTLR...*** menüponttal tudjuk konfigurálni a generálást, a ***Generate ANTLR Recognizer*** menüponttal pedig generálhatunk. Alapértelmezetten a ***gen*** mappába kerülnek a generált fájlok. Itt nem részletezzük őket, de érdemes átnézni, hogy miket generál az ANTLR (lexer, parser, visitor, listener).

Állítsuk be a konfigurációt az alábbihoz hasonlóan, majd generáltassuk le az ANTLR-el a lexert, parsert, listenert, és visitort:



Ha tesztelni (debugolni) szeretnénk a nyelvtant, akkor jobb klikkeljünk valamelyik szabályon (pl. a kezdőszabályon - *chessVariant*) és válasszuk a ***Test Rule chessVariant*** opciót. Ekkor a lent található ***ANTLR Preview*** ablakban tudjuk tesztelni a nyelvtant. Például megadhatjuk a **chess_input.txt** fájlban található minta szöveget, és megnézhetjük, milyen levezetési fát készít hozzá az ANTLR. Az alábbi ábrán nem ez, hanem egy játékos figuráinak megadása látható (*playerDefinition* szabály):



III. SZEMANTIKAI ELEMZŐ

A nyelvtan elkészítése után az első lépés, hogy olvassuk be a **chess_input.txt** fájlból a minta szöveget, és építsük fel belőle a levezetési fát – jelen esetben Java környezetben. Ehhez a **ChessVariantsRunner.java** fájlban hozzuk létre a következő függvényt:

```
public static ChessVariantsParser.ChessVariantContext parseChessVariant(String source) {
    var lexer = new ChessVariantsLexer(CharStreams.fromString(source));
    var parser = new ChessVariantsParser(new CommonTokenStream(lexer));
    return parser.chessVariant();
}
```

A fenti függvény egy tetszőleges string forrásra elvégzi a lexikai elemzést (a generált *ChessVariantsLexer* segítségével), utána pedig a szintaktikai elemzést (a generált *ChessVariantsParser* segítségével). A végén a *chessVariant* függvényhívás a nyelvtanunk kezdőszabályára utal.

A *main* függvényből hívjuk meg a parseChessVariant függvényt:

```
var input = Files.readString(Paths.get("src\\chess\\chess_input.txt"));
var tree = parseChessVariant(input);
```

Az ANTLR támogatást nyújt a levezetési fa bejárásához a **Visitor tervezési minta** segítségével. Az ANTLR által generált fájlok között megtalálható a *ChessVariantsBaseVisitor* osztály, ebből fogjuk a saját visitorainkat leszármaztatni. A szemantikai elemzést két lépésben fogjuk megvalósítani:

- Az első lépésben bejárjuk a fát és felépítjük a szimbólumtáblát, vagyis begyűjtjük a játékos és figura definíciókat
    - Itt akár végezhetnénk egyszerűbb ellenőrzéseket is (amelyekhez nincs szükség a kigyűjtött játékosokra és figurákra), de ezt most kihagyjuk
    - Már itt kiderülhet az is, hogyha pl. kétszer definiáltunk egy figurát
- A második lépésben ismét bejárjuk a fát, és a szimbólumtábla alapján egyszerű ellenőrzéseket végzünk
    - Jelen esetben annyit, hogy ahol figurákra hivatkozunk, ott létezzen is az adott figura
    - Ezen kívül minden egyes koordinátára ellenőrizzük, hogy a tábla keretein belülre mutat-e
- A hibákat konzolon jelezzük a felhasználónak, megjelölve a hiba pontos helyét (sor, oszlop)

Az első visitor kódja (mely a szimbólumtábla felépítéséért felelős) alább látható:

```
public class ChessVariantsFirstVisitor extends ChessVariantsBaseVisitor<Object> {

    private SymbolTable table;

    public ChessVariantsFirstVisitor(SymbolTable table) {
        this.table = table;
    }

    @Override
    public Object visitPlayerDefinition(ChessVariantsParser.PlayerDefinitionContext ctx) {
        Symbol symbol = new Symbol(ctx.playerName().getText(), "player");
        table.insertSymbol(symbol);

        return super.visitPlayerDefinition(ctx);
    }

    @Override
    public Object visitPieceDefinition(ChessVariantsParser.PieceDefinitionContext ctx) {
        Symbol symbol = new Symbol(ctx.pieceName().getText(), "piece");
        table.insertSymbol(symbol);

        return super.visitPieceDefinition(ctx);
    }
}
```

Érdemes megfigyelni, hogy a visitor minta szerint, minden parser szabály bejárásához (egészen pontosan a generált *Context* osztály bejárásához) egy külön visit függvényt írhatunk. A bejárás mélységi keresés alapján történik (*return super. …),* ezt is meg lehet változtatni, de ezt ritkán szoktuk megtenni. Az ANTLR tehát minden parser szabályból egy saját osztályt generál (ezeket a szintén generált *ChessVariantsParser* osztályban nézhetjük meg).

A második visitor kódja (mely az ellenőrzéseket végzi) a következő:

```
public class ChessVariantsSecondVisitor extends ChessVariantsBaseVisitor<Object> {
    private SymbolTable table;
    private int width = -1;
    private int height = -1;

    public ChessVariantsSecondVisitor(SymbolTable table) {
        this.table = table;
    }

    @Override
    public Object visitDimensions(ChessVariantsParser.DimensionsContext ctx) {
        width = Integer.parseInt(ctx.width().getText());
        height = Integer.parseInt(ctx.height().getText());

        return super.visitDimensions(ctx);
    }

    @Override
    public Object visitCoordinates(ChessVariantsParser.CoordinatesContext ctx) {
        int x = Integer.parseInt(ctx.xCoord().getText());
        int y = Integer.parseInt(ctx.yCoord().getText());

        // 0 indexing
        if (x >= width || y >= height)
            System.out.println("Invalid coordinates at line " + ctx.start.getLine() +
                    ", column " + ctx.start.getCharPositionInLine());

        return super.visitCoordinates(ctx);
    }

    @Override
    public Object visitPiecePlacement(ChessVariantsParser.PiecePlacementContext ctx) {
        var pieceName = ctx.pieceName().getText();
        var symbol = table.lookupSymbol(pieceName);

        if (symbol == null)
            System.out.println("Invalid piece type at line " + ctx.start.getLine() +
                    ", column " + ctx.start.getCharPositionInLine());

        return super.visitPiecePlacement(ctx);
    }
}
```

A hibák jelzését éles alkalmazásban érdemes lenne kiszervezni egy külön függvénybe, amely a kapott Context osztály alapján jelzi (konzolon vagy egyéb módon) a hibát a sor és oszlop megjelölésével. A koordináták ellenőrzése – mint korábban volt róla szó – akár áthelyezhető az első visitorba is.

Végezetül egészítsük ki a *main* függvényt, hogy elvégezze a szemantikai elemzést:

```
// …

SymbolTable table = new SymbolTable();
new ChessVariantsFirstVisitor(table).visit(tree);
System.out.println(table);

new ChessVariantsSecondVisitor(table).visit(tree);
```

Az itt bemutatott szemantikai elemző egy egyszerű megoldás egy (jelenleg) egyszerű nyelv esetén. Egy komplexebb nyelv esetén a szemantikai elemző jellemzően sokkal bonyolultabb, különösen akkor, ha a típusrendszer is szerepet játszik benne. Ezekre a problémákra jelen gyakorlat anyaga nem tér ki.

IV. Opcionális feladatok otthonra

A következő feladatok elvégzése opcionális. A feladatok megoldását el lehet küldeni a tárgy oktatóinak (jelen útmutató írásakor a következő email címre: Somogyi.Ferenc@aut.bme.hu), akik véleményezik a megoldást, de plusz pont vagy egyéb jutalom nem jár érte. A feladatok helyenként szándékosan alul specifikáltak, pl. nem térünk ki külön arra, hogy milyen szemantikai elemzést lenne érdemes elvégezni egy új nyelvi feature esetén!

Játékos figuráinak megadása – intervallum koordináták (könnyű)

Legyen lehetőség egy játékos figuráinak megadásánál intervallum koordinátákat használni!

```
player black (
    pawn [1, 0-7]
    superPawn [2-4, 0-3]
)
```

Körök megadása – játékosok (könnyű)

Támogassuk a körök megadásánál a játékosok szerint megkülönböztetett köröket!

```
turn white (
    // …
)

turn black (
    // …
)
```

Körök megadása – elágazások (közepes)

Támogassuk a körök megadásánál az elágazásokat (~*if statement*, a szintaxis mindegy)! Például:

- Aritmetikai műveletek, modulo operátor (pl. kör sorszámánál)
- Játék helyzete alapján történő elágazás (pl. adott játékos adott figuráinak száma alapján)
- stb.

```
turn (
    turn 10: if (count white.pawn < 4) move 3 piece
       else move 2 piece
    turn X: if (x % 2 == 1) move 2 piece
       else move 1 piece
)
```

Figurák lépéseinek megadása – egyszerűsítő szintaxis (közepes)

Találjunk ki egy egyszerűbb szintaxist a bonyolultabb lépések leírására! Például a huszár lépéseinek megadásánál jelenleg sokat kell írnunk. Továbbra is támogassuk a régi megadási módot is!

Figurák lépéseinek megadása – speciális lépések (közepes)

Különböztessük meg a figurák speciális lépéseit, melyet csak bizonyos feltételek mellett hajthatnak végre! Például a gyalog az első körben kettőt léphet előre, sáncolás, vagy akár saját speciális lépések definiálása.

Körök megadása – speciális lépések (közepes)

Egészítsük ki a körök megadását úgy, hogy bizonyos körökben (vagy feltételek alapján, ha a releváns feladat is elkészült) csak bizonyos lépéseket lehessen végrehajtani! Tipp: érdemes a lépéseket kategóriákba osztani, ld. előző feladat.

Több nyelvre bontás (közepes)

Bontsuk a nyelvet négy külön nyelvre a különböző funkciók alapján! Bónusz: használhatunk egy vagy több közös lexert, amit a *tokenVocab* opcióval tudunk importálni. Gondoljuk végig, mi lesz ekkor a szemantikai elemzés menete!

Játék vége kezelése (nehéz)

Legyen megadható, hogy mikor ér véget a játék! Próbáljuk meg először lefedni a klasszikus sakk szabályait (patt, sakk, sakk-matt)!

Sakktábla megadása – nem csak négyzetes mezők (nehéz)

Legyen lehetőség nem csak négyzetes mezőket tartalmazó táblák megadására!