Elosztott rendszerek laboratórium
Szakterületi nyelv fejlesztése ANTLR segítségével

1 A labor menete

A labor elején egy egyszerű, Java-ban megírt 2D-s játék vezérlésére hozunk létre egy szakterületi nyelvet. A labor második része önálló feladatokból áll, mely során a hallgatók kiegészítik a közösen elkészített nyelvet, illetve létrehoznak egy új nyelvet, amely segítségével a játékhoz pályák készíthetők, egyszerű és kényelmes módon.

A feladatok megoldása során **jegyzőkönyv** készítése szükséges. A jegyzőkönyv egy PDF fájl legyen, amibe mindig csak annyit másoljunk be, amennyit az adott feladat kér (választhatóan **screenshot vagy kódrészlet** formájában), egyéb információk (pl. NEPTUN kód, stb.) nem kellenek. A **vezetett** feladatok összesen **elégséges (2)** osztályzatot eredményezhetnek. Minden további **önálló** feladat helyes megoldása **+1 jegyet** jelent. A labor teljesítéséhez **IntelliJ IDEA**[1] fejlesztőkörnyezet szükséges.

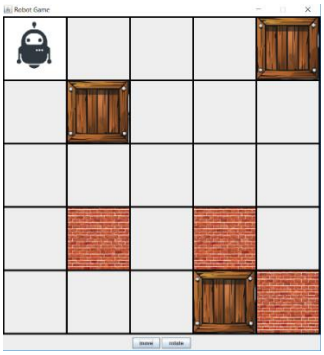
2 ANTLR

Az ANTLR (Another Tool for Language Recognition[2]) egy népszerű parser generátor, mely segítségével egyszerűen tudunk saját (szakterületi) nyelveket készíteni. Az ANTLR elvégzi helyettünk a lexer és parser generálását, amiket kézzel nehéz és fáradtságos lenne megírni. Több népszerű célnyelvet támogat, pl. Java, C#, Python, JavaScript, stb. Az ANTLR jól követi a fordítóprogramok klasszikus lépéseit, különös tekintettel a lexikai, szintaktikai, és szemantikai elemzésre.

3 Vezetett feladat

3.1 A projekt előkészítése

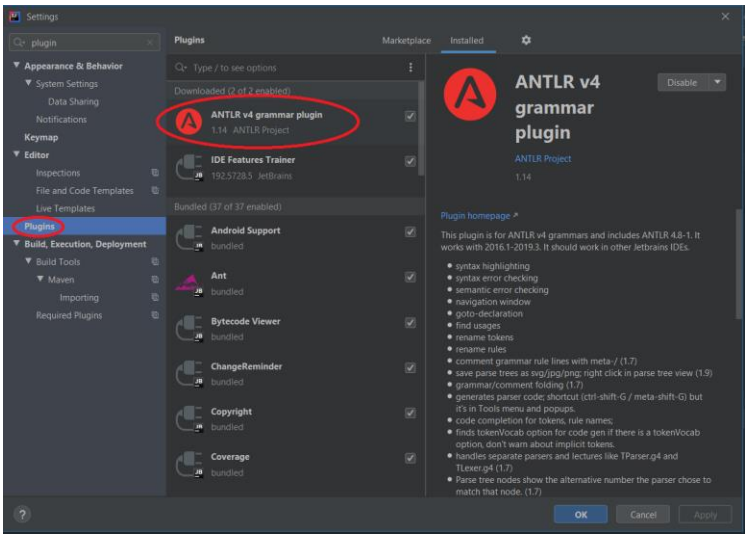
A játékban egy robotot irányítunk, a cél a pályán lévő összes láda megszerzése. A robot két elemi műveletet képes végrehajtani: 1) jobbra fordul 90 fokot, 2) lép egyet az adott irányba. A robot kezdetben mindig lefele néz. Falra a robot nem tud lépni. A vezetett feladat során célunk egy szakterületi nyelv kifejlesztése a robot vezérléséhez.



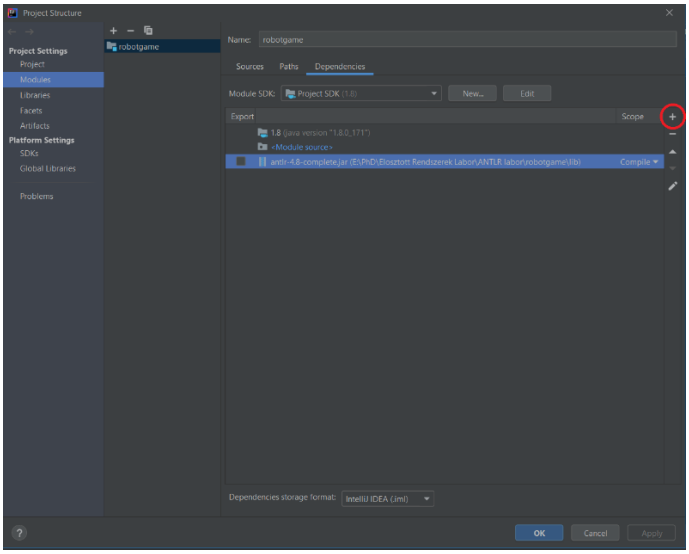
[1] https://www.jetbrains.com/idea/download/ (a Community edition ingyenes)
[2] https://www.antlr.org/

Indítsuk el az IntelliJ IDEA-t, és **importáljuk be** (Next, Finish…, >= Java 8) a tárgy oldalán található projektet (***ANTLR labor – kiinduló projekt***), miután azt kicsomagoltuk. A projekt egy egyszerű Java Swing-es alkalmazás, a Model-View-Controller architektúrát követi. Nézzük meg a projektben található osztályokat! A labor során a *View* osztályt nem módosítjuk, ezt röviden fussuk át; a ***Model***-t és ***GameController***-t részletesebben nézzük meg! A *GameController* osztály publikus függvényeire akár tekinthetünk úgy is, mint egy keretrendszer publikus API-jára, amit a nyelvünk használhat. Próbáljuk ki a játékot a *GameController* futtatásával: jobb klikk > Run 'GameController.main()'!

Mielőtt elkezdenénk a nyelv fejlesztését, győződjünk meg róla, hogy az **ANTLR plugin telepítve** van: File > Settings > Plugins, ld. az alábbi ábrán. A plugin számos kényelmi funkciót nyújt: testreszabott szerkesztő a nyelvtan fájloknak, lexer és parser kényelmes generálása (ezt parancssorból is megtehetnénk), szintaxisfák „grafikus" megjelenítése, stb.



Ahhoz, hogy Java kódból használni tudjuk az ANTLR-t, szükségünk lesz az **ANTLR runtime**-ra is. Ezt JAR formájában, függőségként kell hozzáadni a projekthez: *lib/antlr-4.8-complete.jar.* Győződjünk meg arról is, hogy a függőség helyesen szerepel itt: File > Project Structure > Modules > Dependencies. Ha nem lenne hozzáadva, adjuk hozzá a lenti módon:



3.2 A szakterületi nyelv létrehozása

Az *input/sample.robot* fájl az elkészítendő nyelvünkhöz tartalmaz **példakódot**. Nézzük ezt meg! Kezdetben két utasítást szeretnénk megvalósítani, a robot elemi műveleteit:

- move: ennek hatására a robot 1 mezőnyit mozog
- rotate: ennek hatására a robot 90 fokot fordul jobbra

Hozzunk létre egy új package-et (*language.controller*) az *src* mappa alá. A package-en jobb klikk > New File > „RobotController.g4". A g4 az **ANTLR nyelvtan** formátuma, a 4 a verziószámra utal. Ha mindent jól csináltunk, akkor az IntelliJ automatikusan hozzárendeli az ANTLR plugint a fájlhoz és megkapjuk a szerkesztő funkciókat. A nyelvtan kötelezően a fájl nevével megegyező kell, hogy legyen, ezért kezdjük a következő sorral:

```
grammar RobotController;
```

Az ANTLR kétfajta szabályt különböztet meg:

- A **lexer szabályok** a lexikai elemzés folyamatát szabályozzák. Itt adjuk meg, hogy milyen elemi egységekből (tokenekből) épül fel a nyelvünk szintaxisa. A szabályok sorrendje számít, az ANTLR egy adott szöveg (pl. „move") illesztése során sorban próbálja illeszteni a szöveget a szabályokra. Konvenció szerint csupa nagybetűvel írjuk a szabályok nevét.
- A **parser szabályok** a szintaktikai elemzés folyamatát szabályozzák. Itt adjuk meg, hogy hogyan épül fel a nyelvünk szintaxisa, milyen utasításaink és konstrukcióink vannak, azok hogyan épülnek egymásra. A szabályok sorrendje nem számít, kivéve az első szabályt, ami a nyelvünk kezdőszabálya lesz. Konvenció szerint camelCase-el írjuk a szabályok nevét.

Bottom-up megközelítés alapján először írjuk meg a **lexer szabályokat**! Eleinte a nyelv egyszerű lesz, az alábbi tokenekre van csak szükségünk:

```
MOVE: 'move';
ROTATE: 'rotate';
EOS: ';';
```

A whitespace karakterekből és kommentekből nem szeretnénk, hogy a nyelv részét képező token keletkezzen, ezért ezeket speciálisan kezeljük (a „skip" akció pontosan ezt csinálja):

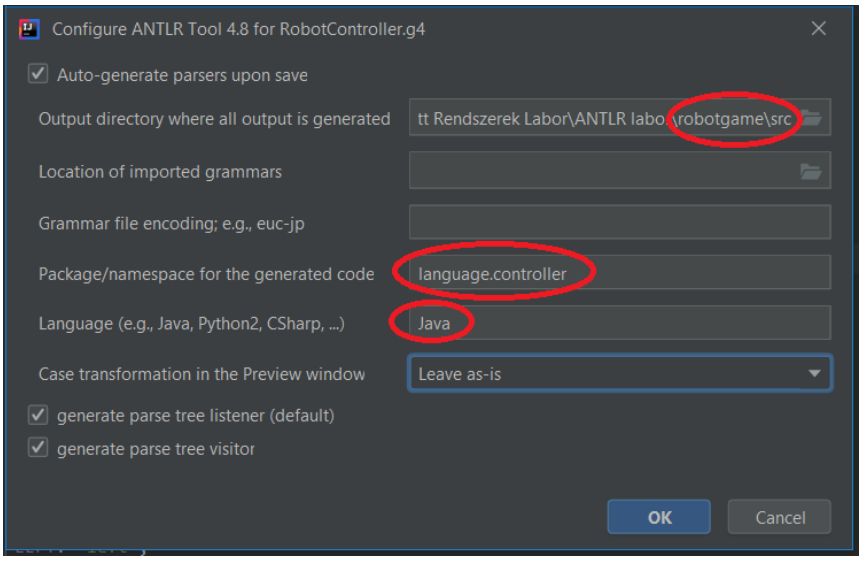
```
WS: (' '| '\t' | '\n' | '\r') -> skip;
COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
```

Ezután a **parser szabályok** következnek (konvenció alapján tegyük őket a lexer szabályok fölé – vagyis a fájl elejére):

```
 program: statement+;
statement: moveStatement | rotateStatement;
moveStatement: MOVE EOS;
rotateStatement: ROTATE EOS;
```

A fenti szabályok azt írják le, hogy a programunk utasítások sorozatából áll. A '+' miatt egy vagy több utasításunk lehet, hasonlóan a reguláris kifejezésekhez. Használhatunk még '*' (0 vagy több) és '?' (0 vagy 1) karaktereket is. Egy utasítás csak *move* vagy *rotate* lehet, mindig pontosvesszővel lezárva.

Teszteljük a nyelvtant: jobb klikk valamelyik parser szabályon (pl. program) > Test rule… Jobb oldalt a beírt szöveghez megjelenik a megfelelő szintaxisfa, a megfelelő hibákkal, amennyiben a szöveg hibás. Ezután **állítsuk be**, hogy hova és milyen fájlok (lexer, parser, stb.) generálódjanak: jobb klikk a nyelvtan fájlon belül > Configure ANTLR…, és adjuk meg az alábbi konfigurációt. Majd jobb klikk a nyelvtan fájlon belül > Generate ANTLR Recognizer paranccsal **generáljuk le** a fájlokat.



Nézzük meg, hogy milyen fájlok **generálódtak**! A *Lexer* és *Parser* osztályok és interfészek a lexikai és szintaktikai elemzésért felelősek (egy adott szöveg alapján tokenizálnak, illetve felépítik a szintaxisfát), a *Listener* és *Visitor* osztályok pedig az éppen készülő, vagy pedig az elkészült szintaxisfa bejárásáért felelősek. Természetesen az elkészült szintaxisfát tetszőleges módon is bejárhatjuk, a visitor minta használata egy best practice jellegű megoldást ad.

Hozzuk létre a **MyRobotControllerVisitor** osztályt a *language.controller* package-en belül (jobb klikk > New > Java Class):

```
public class MyRobotControllerVisitor extends RobotControllerBaseVisitor<Object> {
   private GameController controller;

   public MyRobotControllerVisitor(GameController controller) {
      this.controller = controller;
   }
}
```

A visitor mintát követve, a megfelelő visitXY függvények felüldefiniálásával oldjuk meg a move és rotate **parancsok feldolgozását** (minden parser szabályhoz egy *Context* osztály tartozik):

```
@Override
public Object visitMoveStatement(RobotControllerParser.MoveStatementContext ctx) {
      controller.move();
      return super.visitMoveStatement(ctx);
}
```

```
@Override
public Object visitRotateStatement(RobotControllerParser.RotateStatementContext ctx) {
    controller.rotate();
    return super.visitRotateStatement(ctx);
}
```

Kössük be a visitort a *GameController* osztályba az alábbi kódrészlet alapján! Ügyeljünk rá, hogy az *org.antlr.v4...* csomagbeli importokat használjuk! Frissítsük a *move* és *rotate* függvényeket is, hogy a parancsok ne azonnal, hanem fél másodpercenként hajtódjanak végre!

```
private GameController() {
    …
    parseControllerFile();
}

private void parseControllerFile() {
    try {
      File file = new File("res\\sample.robot");
      InputStream fileStream = new FileInputStream(file);
      ANTLRInputStream inputStream = new ANTLRInputStream(fileStream);

      RobotControllerLexer lexer = new RobotControllerLexer(inputStream);
      CommonTokenStream tokens = new CommonTokenStream(lexer);

      RobotControllerParser parser = new RobotControllerParser(tokens);
      RobotControllerParser.ProgramContext tree = parser.program();

      MyRobotControllerVisitor visitor = new MyRobotControllerVisitor(this);
      visitor.visit(tree);
    }
    catch (IOException ignored) { }
}

public void move() {
    model.movePlayer();
    view.refresh();
    try { Thread.sleep(500); }
    catch (InterruptedException ignored) {}
    checkWinCondition();
  }

  public void rotate() {
    model.rotatePlayer();
    view.refresh();
    try { Thread.sleep(500); }
    catch (InterruptedException ignored) {}
  }
```

A *parseControllerFile* függvény tartalmazza a lexikai és szintaktikai elemzés lépéseit: 1) létrehoz egy lexert, 2) tokenizál, majd 3) létrehoz egy parsert és 4) felépíti a szintaxisfát. Ezek után példányosítja a visitort és átadja neki a *GameController* objektumot. A visitorral tulajdonképpen egy **interpretert** valósítottunk meg, ami futási időben vezérli a robotot, vagyis adja ki a megfelelő parancsokat a bemeneti szöveg alapján. **Próbáljuk ki** a nyelvet, futtassuk a *sample.robot* fájlban található utasításokat (változatlanul futtatva a *GameController main* függvényét)!

JEGYZŐKÖNYV: a RobotController.g4 fájl teljes tartalma

3.3 A szakterületi nyelv továbbfejlesztése

Egészítsük ki a nyelvet, hogy támogassa a **ciklusok** használatát! Legyen lehetőség a **log** parancs kiadására is, ami a paraméterként kapott stringet egy dialógusablakban kiírja a képernyőre!

A ciklusok és a logolás szintaxisa az alábbi legyen:

```
loop (4) {
    move;
    rotate;
    move;
    rotate;
    move;
    log("loop done");
}
```

Szükségünk lesz **új tokenekre** a kétféle zárójeltípus, valamint a *loop* és *log* kulcsszavak miatt. Ezenkívül támogatnunk kell a stringeket is, amit a legegyszerűbben úgy oldhatunk meg, hogy két idézőjel között, az „új sor karakterek” kivételével bármi állhat. Tehát vegyük fel a nyelvtanba a következő lexer szabályokat:

```
LOOP: 'loop';
LOG: 'log';
LPAREN: '(';
RPAREN: ')';
LCURLY: '{';
RCURLY: '}';

INT: [0-9]+;
STRING: '"' (~[\r\n])* '"';
```

Egészítsük ki a **parser szabályokat**, hogy támogassuk a két újfajta utasítást. Az *amount* és *logMessage* szabályok csak a szebb struktúra miatt került be a nyelvtanba, szemantikai jelentősége nincs, használhattuk volna az INT és STRING tokeneket is önmagukban.

```
statement: moveStatement | rotateStatement | loopStatement | logStatement;

loopStatement: LOOP (LPAREN amount RPAREN) LCURLY statement+ RCURLY;
amount: INT;
logStatement: LOG (LPAREN logMessage RPAREN)? EOS;
logMessage: STRING;
```

Generáljuk újra a nyelvtant (jobb klikk a nyelvtan fájlon belül > Generate ANTLR Recognizer)!

Végül **egészítsük ki a visitorunkat**, hogy támogassa a *loop* és *log* parancsokat:

```
@Override
public Object visitLoopStatement(RobotControllerParser.LoopStatementContext ctx) {
   Object result = null;
   for (int i = 0; i < Integer.parseInt(ctx.amount().INT().getText()); i++) {
      for (RobotControllerParser.StatementContext stm : ctx.statement()) {
         result = visit(stm);
      }
   }
   return result;
}

@Override
public Object visitLogStatement(RobotControllerParser.LogStatementContext ctx) {
   controller.displayMessage(ctx.logMessage().STRING().getText());
   return super.visitLogStatement(ctx);
}
```

Figyeljük meg, hogyan képezi le a **kód a nyelvtan struktúráját**, például az *amount*, *logMessage*, *INT* és *STRING* függvényeken keresztül! A *getText* függvénnyel az adott kontextushoz (vagyis szabályhoz) tartozó nyers szöveget kérhetjük le.

Próbáljuk ki a nyelvet a *sample.robot* fájl átírásával (pl. használhatjuk a fenti, szintaxist demonstráló példakódot) és a *GameController* futtatásával!

JEGYZŐKÖNYV: screenshot a játékról futtatás közben (log ablak látszódjon)

Végezetül érdemes megemlíteni, hogy a laboron a **szemantikai elemzéssel** nem foglalkozunk. Ez azt jelenti, hogy bár a nyelvben szintaktikailag helytelen kódot nem adhatunk meg, szemantikailag helytelen kód még előfordulhat. Például tetszőlegesen nagy ismétlésszámú ciklust megadhatunk, ami túl sok ideig futna. A vezetett feladatoknál ez nem okoz nagy problémát, viszont az önálló feladatoknál (például a pályaszerkesztő nyelv esetén a pálya keretein kívül lévő koordináták megadása esetén) feltűnőbb lehet a szemantikai elemzés hiánya. Mivel a labor keretein belül nem valósítunk meg szemantikai elemzést, ezért a helytelen kód ellenőrzésével most nem kell foglalkoznunk. ANTLR esetén ez tipikusan egy különálló, ellenőrző visitor segítségével valósulna meg, a kódgenerálás (jelen esetben az interpretálás) előtt.

4 Önálló feladatok

4.1 Move és rotate parancsok továbbfejlesztése (+1 jegy)

Egészítsük ki a robotvezérlő nyelvet úgy, hogy a *move* és *rotate* parancsok paraméterezhetőek legyenek! Maradjon meg a régi változat is, tehát lehessen paraméter nélkül is használni őket! A *move* esetén a paraméter a lépések száma, míg a *rotate* esetén a célirány legyen! Ne felejtsük el a visitort se kiegészíteni! Kövessük az alábbi szintaxist:

```
rotate(right);
move(4);
```

TIPP: a *rotate* parancsnál a visitorban használjuk a *GameController* osztály *getPlayerFacing* függvényét!

JEGYZŐKÖNYV: a nyelvtan kiegészítései és a visitor kiegészítései

4.2 Pályaszerkesztő nyelv elkészítése (+1 jegy)

Hozzunk létre egy új nyelvtant (**RobotLevelMaker.g4**) a szintén újonnan létrehozandó *language.levelmaker* package alá. Hasonlóan a robotvezérlő nyelvhez, ne felejtsük el konfigurálni az ANTLR (lexer, parser, visitor, stb.) generálását! Készítsük el a nyelvet a következők szerint:

- Az első utasítás mindig a **DIM[X][Y];** utasítás, ahol X és Y a pálya méreteit írja le.
- A **PLAYER[X][Y]** utasítással a játékos pozícióját adhatjuk meg.
- A **WALL[X][Y]** utasítással egy fal pozícióját adhatjuk meg.
- A **CRATE[X][Y]** utasítással egy láda pozícióját adhatjuk meg.
- Az üres mezőket nem kell megadnunk, azok automatikusan kitöltődnek.
- Írjuk át a *GameController* osztályt, hogy a pályát leíró fájl (pl. *input/level.robot*) alapján hozza létre a *Model*-t! Ehhez a *Model*-en is változtatnunk kell!
- Használjunk visitort a nyelv feldolgozásához! A fontos információkat eltárolhatjuk a visitoron belül, publikus attribútumok formájában!

JEGYZŐKÖNYV: a teljes nyelvtan és visitor, valamint a *GameController* módosított részei

4.3 Pályaszerkesztő nyelv továbbfejlesztése (+1 jegy)

Egészítsük ki a pályaszerkesztő nyelvet úgy, hogy a falak lehelyezésénél támogassa az ún. „intervallum koordináták” használatát! Intervallum koordináta megadása esetén az adott intervallum minden koordinátájára kerüljön fal objektum! Kövessük az alábbi szintaxist (ld. *input/level.robot* – kikommentezett sor):

```
WALL[0-3][3];
```

TIPP: a visitorban érdemes egy-egy listában tárolni az adott falhoz tartozó koordinátákat!

JEGYZŐKÖNYV: a nyelvtan kiegészítései és a visitor kiegészítései