



# MODELLALAPÚ SZOFTVERFEJLESZTÉS – 2. GYAKORLAT – ANTLR

Lexikai és szintaktikai elemzés

Dr. Somogyi Ferenc

## Szerzői jogok

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Modellalapú szoftverfejlesztés c. tantárgyat felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármely eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző előzetes engedélyével történhet.



## I. BEVEZETÉS

A gyakorlat célja, hogy a gyakorlatban is megismerkedjünk a klasszikus fordítás első két 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
- Egy prototípus programozási nyelv (TinyScript) kezdetleges implementációja a fentiek alapján

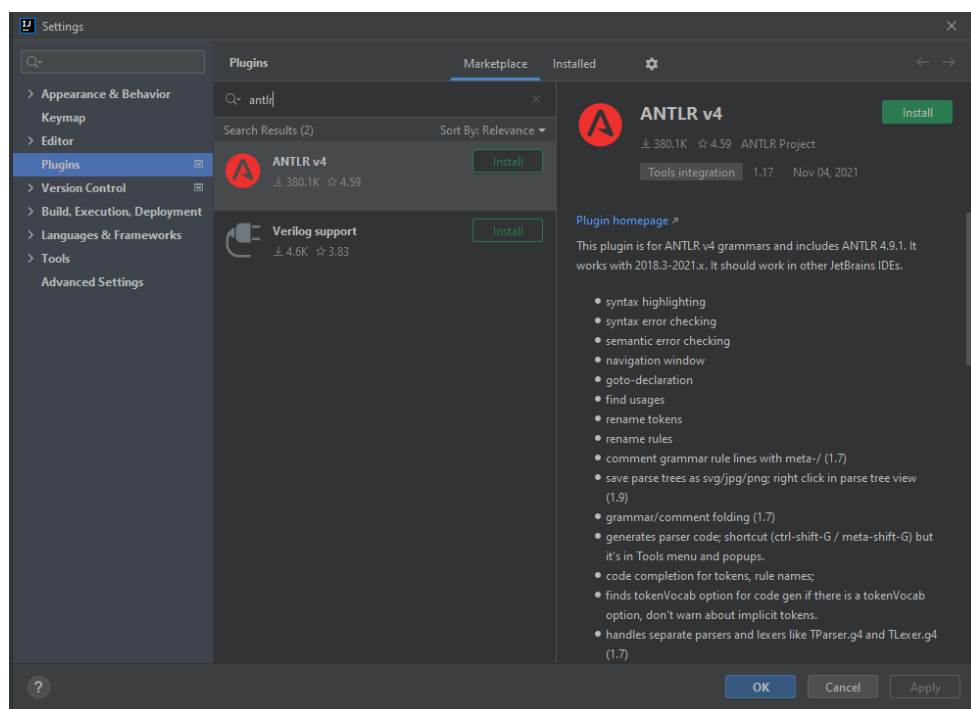
Fontos: jelen útmutató (a későbbiekkel együtt) az **ANTLR 4**-es verziójához készült. Az **ANTLR 3**-as verziója sok szempontból hasonló a 4-eshez, viszont még több szempontból nem. Az interneten fellelhető segédanyagoknál és kérdéseknél (pl. stackoverflow) érdemes meggyőződni róla, hogy melyik verzióról 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 feltelepíteni az ANTLR plugint – különben pl. a lexer és parser generálást parancssorból 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). Jelen útmutatóban Java-t és IntelliJ-t használunk, de választhatnánk más is.

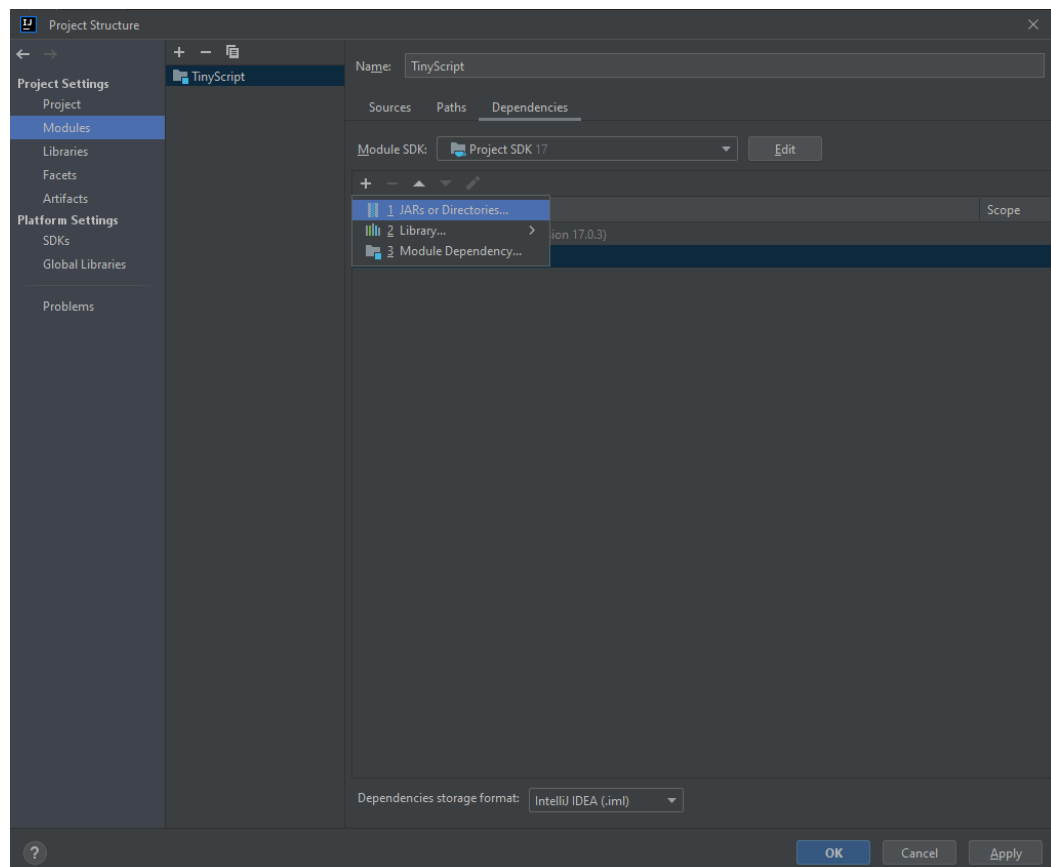
---

Ezután nyissuk meg a kiinduló projektet (**TinyScript**)! A **lib** mappa alatt megtalálható az ANTLR runtime (egy *jar* fájl), ami a runtime funkciók eléréséhez (pl. szintaxisfa felépítése futási időben, szemantikai elemzés elvégzése stb.) szükséges. A *tinyscript* és *tinyscript.exceptions* package-ben négy fájl található:

- **input.tys**: egy példa bemeneti szöveget tartalmaz; ezt fogjuk később feldolgozni
- **TinyScriptRunner.java**: egy *main* függvényt tartalmaz; itt fogjuk később használni a runtime jar-t és feldolgozni a bemeneti szöveget a generált lexer és parser segítségével
- **TinyScriptException.java** és **TinyScriptExceptionHandler.java**: egy exception a hiba pontos helyét is tartalmazza (sor, oszlop); az összes exception-t gyűjtjük, ahelyett, hogy csak az első hibáig futna a program

Amennyiben új projektet hozunk létre, akkor a runtime *jar* fájlt kézzel (vagy 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 fájlstruktúrájába (pl. **lib** 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. A TINYSCRIPT NYELV SPECIFIKÁCIÓJA

A TinyScript egy prototípus programozási nyelv, amely néhány alapvető, programozási nyelveknél megszokott funkciót támogat. A nyelvnek nem célja, hogy teljes értékű programozási nyelv legyen, ugyanis ehhez nagyon sokat kellene dolgoznunk, ami nem fér bele a gyakorlat kereteibe. Viszont a nyelven keresztül betekintést nyerhetünk a fordítók világába a gyakorlatban, vagyis abba, hogy egyes, programozásból megismert koncepciókat hogyan tudunk leírni és feldolgozni. Jelen gyakorlat végére elkészítjük a TinyScript nyelv szintaxisát, vagyis képesek leszünk a nyelven leírt kódot beolvasni, a szintaktikai hibákat kiszűrni. A szemantikai elemzés és kódgenerálás egy későbbi gyakorlaton kerülnek elő.

A TinyScript nyelvben függvényeket definiálhatunk, valamint egy *main* blokkon belül a program futását írhatjuk le. Függvényeket definiálni a *main* blokk előtt és után is lehetséges. A nyelv a következő funkciókat támogatja:

- **main** blokk: **tetszőleges utasítások** szerepelhetnek a blokkon belül
- **függvény** definíció: a függvénynek van **neve**, **visszatérési típusa**, **paraméterei** (opcionális) és **törzse**
- a primitív típusok közül a nyelv csak a következőket támogatja: **int**, **string**, **bool**
  - a **null** érték is előfordulhat, tegyük fel, hogy a **string nullable**, más típus nem az (de később bevezethetnénk osztályokat, amik szintén nullable kell, hogy legyenek)
- **változó deklarálása**
  - **var** kulcsszóval vagy **típus** megadásával
  - **kezdeti értéke** is lehet, var kulcsszó esetén kötelezően kell, hogy legyen kezdeti érték
- **változó értékadás**: tetszőleges kifejezést értékül adhatunk egy létező változónak
- **while** ciklus: feltétel és törzs
- **if** utasítás: feltétel, törzs, opcionálisan **else** ága is lehet
- **függvényhívás**
  - a paramétereket helyesen kell átadni
- visszatérés (**return** utasítás): csak függvény törzsén belül szerepelhet, main blokkon belül nem
- kifejezések (**expression**)
  - a következőkben szerepelhet: változó deklarálás, változó értékadás, függvényhívás paramétere, while és if feltétel, return utasítás
  - **literal** értékek (null, int, string, bool), változó **referencia** név alapján
  - **zárójelezés**, **függvényhívás**
  - **aritmetikai** (+, -, \*, /) és **logikai** operátorok (>=, <=, >, <, ==, !=)
- **kommentek**: egy és többsoros

Ha szeretnénk példát látni a nyelv szintaxisára, érdemes a kiinduló projektben található **input.tys** fájlt megnézni.

### III. A NYELVTAN MEGÍRÁSA

A **tinyscript** packageen **jobb klikk --> New --> File**, a nyelvtan neve legyen **TinyScript.g4**. A g4 az ANTLR fájlformátuma, a „g” a grammar szóra utal, míg a „4” az ANTLR fő verziójára. A következőkben először általánosságban nézzük meg egy ANTLR nyelvtan felépítését, majd megírjuk a TinyScript nyelvtanát.

#### ANTLR NYELVTANOKRÓL ÁLTALÁNOSSÁGBAN

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

```
grammar TinyScript;
```

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 (*terminális szimbólumok*) állhat, aposztrófok között. A bal oldalon a szabály neve szerepel (*nemterminális szimbólum*), 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 (*terminális és nemterminális szimbólumok*). Nyers szöveg is lehet, de a best practice elvek szerint érdemes minden ilyen lexer szabályba kiszerveznünk. Ennek kódszervezési és performance okai is vannak. A bal oldalon a szabály neve szerepel (*nemterminális szimbólum*). A bal oldalt hagyományosan camelCase szerint írjuk.

A szabályok megadásánál a reguláris kifejezések világából ismerős quantifier-eket (?, \*, +, stb.) használhatjuk, az EBNF jelöléshez hasonlóan. 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 (token) ne kerüljön bele a levezetési fába, azt a parser ugorja át.

A nyelvtan elkészítésénél kétféle módon gondolkodhatunk. A *bottom-up* módszer szerint először a nyelv alap építőköveit (lexer szabályok) készítjük el, és utána adjuk meg a nyelv szerkezetét (parser szabályok). A *top-down* módszer szerint először a nyelv szerkezetét adjuk meg (parser szabályok), majd utána pontosítjuk, hogy az egyes tokenek hogy néznek ki (lexer szabályok). Bármelyik módszert alkalmazhatjuk, akár keverve is a kettőt.

Egy ANTLR nyelvtan fájlban tipikusan először a parser szabályok szerepelnek, majd utána következnek a lexer szabályok, függetlenül attól, hogy melyeket írtuk meg előbb. Ennek funkcionális oka nincs, csupán konvenció szerint a legtöbb nyelvtanban így találhatók meg a szabályok, így érdemes nekünk is ezt követnünk.

---

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 TINYSCRIPT NYELVTANÁNAK ELKÉSZÍTÉSE

A nyelvtant most *bottom-up* módszerrel írjuk meg, először a lexer szabályok, majd a parser szabályok következnek. Kezdjük a TinyScript nyelvben használatos kulcsszavakkal, operátorokkal, valamint egyéb speciális karakterekkel (pl. zárójelek, pontosvessző).

```
LPAREN: '(';
RPAREN: ')';
LCURLY: '{';
RCURLY: '}';
EOS: ';';
COMMA: ',';

EQ: '==';
NEQ: '!=';
NEG: '!';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';

ASSIGN: '=';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';

IF: 'if';
ELSE: 'else';
WHILE: 'while';
VAR: 'var';
MAIN: 'main';
RETURN: 'return';
VOID: 'void';
```

A literal értékek leírására a következő parser szabályokat írhatjuk meg. Az INT szabályt pontosíthatjuk a nyelvtanban (hogy ne kezdődhessen 0-val), viszont az ilyen speciális követelményeket (amennyiben fontosok egyáltalán a nyelv szempontjából) tipikusan inkább a szemantikai elemzésnél ellenőrizzük a nyelvtan túlbonyolítása helyett.

```
NULL : 'null';
TRUE: 'true';
FALSE: 'false';
STRING: '"' (~[\r\n])* '"';
INT: [0-9]+;
```

Végezetül írjuk meg azokat az általánosabb lexer szabályokat (ezek nagy részét – kicsit más formában – előadáson is láthattuk), amelyek több helyen is kellenek később.

```
ID:      [a-zA-Z][a-zA-Z0-9_]*;
WS:      (' ' | '\t' | '\n' | '\r') -> skip;
COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\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 (esetünkben a fentieket) mindig tegyük a fájl végére!

Ezek után a parser szabályok következnek, ezeket ne felejtjük konvenció szerint a fájl elejére tenni. A **kezdőszabály** (*program*) leírja a nyelv szerkezetét, vagyis, hogy a *main* blokk előtt és után függvény definíciók lehetnek.

```
program
    :      functionDefinition* main functionDefinition*
    ;
```

---

*A parser (és lexer) szabályok formázása a .g4 fájlban nem számít, a funkcionalitás ugyanaz marad.*

---

A **függvények definiálásáért** a következő parser szabályok felelnek. Függvény paraméterből akármennyi lehet (vesszővel elválasztva), ez a rész opcionális is. A *parameterType* és hasonló szabályok (ahol a jobboldalon csak ID található) a strukturáltság miatt kellene, így sokkal könnyebb feldolgozni és olvasni a szintaxisfát, mintha mindenhol csak ID szerepelne. Ez utóbbi esetben ugyanis emlékeznünk kellene például a paraméter típusának és nevének sorrendjére a feldolgozás során. A *returnType* szabálynál a *VOID* token azért van külön kiemelve, mert visszatérési típusnál használható, viszont más helyeken (pl. paraméter típusánál) nem. Mivel a *VOID* lexer szabály előbb van a fájlban, mint az *ID*, ezért mindig az előbbire illeszt a lexer, ha a 'void' szöveggel találkozunk.

```
functionDefinition
    :      returnType
        functionName
        LPAREN (functionParameter (COMMA functionParameter)*)? RPAREN
        LCURLY statement* RCURLY
    ;
returnType: VOID | ID;
functionName: ID;

functionParameter
    :      parameterType parameterName
    ;
parameterType: ID;
parameterName: ID;
```

A **main blokk** a következőképpen épülhet fel. Egy **utasítás** a specifikációban szereplő bármelyik utasítás lehet. Fontos kiemelni, hogy bár a *returnStatement* része a *statement* szabálynak (vagyis ő maga is egy *statement*), visszatérés értelemszerűen csak függvény törzsében szerepelhet. Írhatnánk olyan szabályt is, ami ezt kezeli, viszont ez elbonyolítaná a nyelv struktúráját, ezért ezt a követelményt célszerűbb a szemantikai elemzés során ellenőrizni.

```
main: MAIN LCURLY statement* RCURLY;

statement
    :      variableDeclaration
    |      assignmentStatement
    |      whileStatement
    |      ifStatement
    |      functionCall EOS
    |      returnStatement
    ;
```

**Változók deklarálásáért és értékadásáért** felelnek a következő szabályok. Deklarálás során a kezdeti érték megadása opcionális, hogy ez mikor kötelező (*var* kulcsszó esetén), azt is a szemantikai elemzés során nézzük majd meg. Az *expression* szabály egy tetszőleges kifejezést takar – a specifikációban leírtak szerint –, ezt később részletezzük.

```
variableDeclaration
    :      (VAR | typeName) varName (ASSIGN expression)? EOS
    ;
typeName: ID;
varName: ID;

assignmentStatement
    :      varName ASSIGN expression EOS
    ;
```

A **while** ciklust és **if** utasítást a következő szabályok írják le. A feltétel azért lett kiemelve egy külön szabályba (*condition*), mert a szemantikai elemzés során egységesebben ellenőrizhető, hogy mindig *bool* típusúnak kell lennie. A *dangling else* problémát itt úgy oldjuk meg, hogy mindig megköveteljük a szeparáló karakterek (jelen esetben kapcsos zárójelek) használatát.

```
whileStatement
    :      WHILE LPAREN condition RPAREN LCURLY statement* RCURLY
    ;
condition: expression;

ifStatement
    :      IF LPAREN condition RPAREN LCURLY statement* RCURLY elseStatement?
    ;
elseStatement
    :      ELSE (ifStatement | LCURLY statement* RCURLY)
    ;
```

---

*Megoldhatnánk úgy is a dangling else problémát, hogy determinisztikusan mindig vagy a külső, vagy a belső if ághoz tartozik egy else ág. Itt érdemes a ?? operátornak utánanézni, bár ennek használata a gyakorlatban nem ajánlott, mert rosszabb a teljesítménye a non-greedy működése miatt.*

---

**Függvényhívás** (*functionCall*) szerepelhet önálló utasításként, illetve később láthatjuk, hogy egy kifejezés részeként is. A hívás során átadott paraméterek tetszőleges kifejezések (*expression*) lehetnek. Egy **return** utasítás szintén tetszőleges kifejezéssel térhet vissza, a típushelyesség ellenőrzése a szemantikai elemzés feladata.

```
functionCall
    :      functionTarget LPAREN parameterList? RPAREN
    ;
functionTarget: ID;

parameterList
    :      (expression (COMMA expression)*)
    ;

returnStatement
    :      RETURN expression EOS
    ;
```



Már csak a **kifejezések** definíciója maradt hátra. Az ANTLR újabb verziói támogatják a közvetlen balrekurziót (*self left recursion*), így egyszerűen definiálhatók a különböző operátorok és precedenciáik. A labelék segítségével (pl. `#primaryExpression`) a különböző kifejezés típusokat jobban el tudjuk különíteni, a generált kódban saját osztály fog generálódni minden label után. A precedenciát fentről lefelé definiáljuk (erősebbtől a gyengébb felé), érdemes különböző példákkal tesztelni az *expression* szabályt és megnézni, hogy milyen fa generálódik belőlük. A fa „természetes” kiolvasása során egyben a precedenciát is megkapjuk.

```
expression
:   primary                                #primaryExpression
  | expression mulDivOp expression         #mulDivExpression
  | expression addSubOp expression         #addSubExpression
  | expression logicalOp expression        #logicalExpression
;

addSubOp
: PLUS | MINUS
;

mulDivOp
: MUL | DIV
;

logicalOp
: GTE | LTE | GT | LT | EQ | NEQ
;

primary
: parenthesizedExpression
| functionCall
| literalExpression
;

parenthesizedExpression
: LPAREN expression RPAREN
;

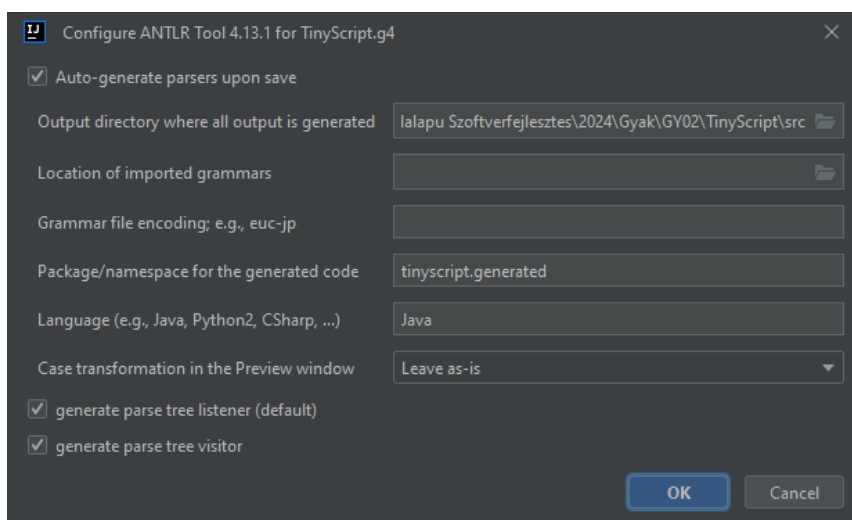
literalExpression
: nullLiteral
| intLiteral
| stringLiteral
| boolLiteral
| varRef
;

varRef: ID;
nullLiteral : NULL;
intLiteral : MINUS? INT;
stringLiteral: STRING;
boolLiteral: TRUE | FALSE;
```

## IV. LEXER ÉS PARSER GENERÁLÁSA, A NYELVTAN TESZTELÉSE

A nyelvtanból az ANTLR képes *lexer* és *parser* generálni, amik automatikusan elvégzik helyettünk a lexikai, illetve a szintaktikai elemzést. IntelliJ környezetben, ha a nyelvtan fájlban bárhol jobb klikkelünk, a **Configure ANTLR...** menüponttal tudjuk konfigurálni a generálást, a **Generate ANTLR Recognizer** menüpont pedig végrehajtja a generálást. Alapértelmezetten a **gen** mappába kerülnek a generált fájlok, ezt érdemes felüldefiniálnunk.

Állítsuk be a konfigurációt az alábbihoz hasonlóan (a package nevére különösen ügyelve), majd generáljuk le a fájlokat a **Generate ANTLR Recognizer** menüponttal.



Itt nem részletezzük őket, de érdemes röviden átnézni, hogy milyen fájlokat generál az ANTLR (lexer, parser, visitor, listener). Röviden összefoglalva, a nyelvtanban definiált parser szabályokból mind külön osztály generálódik (pl. *ProgramContext* a kezdőszabályból), amiket később fel tudunk használni a bejárásakor és feldolgozásakor.

Ha tesztelni („debugolni”) szeretnénk a nyelvtant, akkor jobb klikkeljünk valamelyik szabályon (pl. a *program* kezdőszabályon) és válasszuk a **Test Rule program** opciót. Ekkor a lent található **ANTLR Preview** ablakban tudjuk tesztelni a nyelvtant. Például megadhatjuk az **input.tys** fájlban található példakódot, és megnézhetjük, milyen levezetési fát generál belőle az ANTLR. Az alábbi ábrán nem ez, hanem egy függvény definíciója látható (*functionDefinition* szabály):

