



MODELLALAPÚ SZOFTVERFEJLESZTÉS – 4. GYAKORLAT – ANTLR

Szemantikai elemzés és kódgenerálás

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

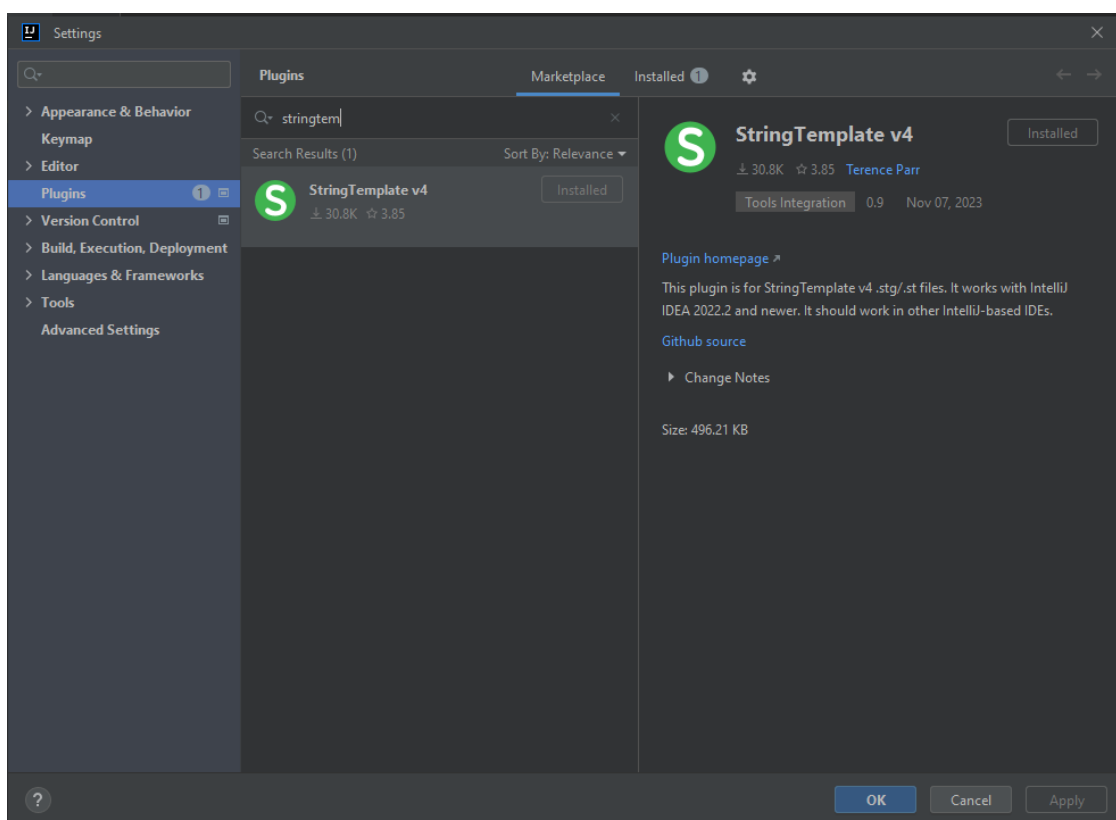
A gyakorlat célja, hogy a 2. gyakorlaton elkészített prototípus programozási nyelv (TinyScript) nyelvtanából kiindulva elkészítsük a nyelv szemantikai elemzőjét, valamint egy kódgenerátort, amely TinyScript nyelven írt bemenetből Java kódot képes generálni. A szemantikai elemző kódja megtalálható a kiinduló projektben (jelen útmutatóban ezt nézzük végig), a **kódgenerátor** egy részét viszont **önállóan** kell majd megírni a **3. házi feladat** keretein belül. A gyakorlat során megismerkedünk a következőkkel:

- Visitor minta használata a szintaxis fa bejárása során
- Szemantikai elemzés: szimbólumtábla és típusellenőrzés megvalósítása
- Sablon alapú kódgenerátor készítése StringTemplate segítségével

Fontos: jelen útmutató 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.

STRINGTEMPLATE BEÜZEMELÉSE INTELLIJ KÖRNYEZETBEN

A StringTemplate¹ egy sablon alapú kódgeneráló eszköz. A kiinduló projektben az ANTLR-hez hasonlóan megtalálható a *lib* mappa alatt egy jar fájl, ami a futáshoz szükséges. Győződjünk meg róla, hogy a jar fájl ott van a projekt függőségei között (**File --> Project Structure... --> Modules --> Dependencies**), és ha nincs, adjuk hozzá. Ezen kívül telepítsük a megfelelő IntelliJ plugint is, szintén az ANTLR-hez hasonlóan. **File --> Settings...** menüpont, válasszuk oldalt a **Plugins**-t, keressünk rá a StringTemplate-re, és telepítsük a plugint (utána indítsuk újra az IntelliJ-t):



¹ <https://www.stringtemplate.org/>

II. SZEMANTIKAI ELEMZÉS

Töltsük le és nyissuk meg a kiinduló projektet! A következőkben a szemantikai elemzés folyamatát fogjuk röviden áttekinteni a kiinduló kód alapján; a teljes szemantikai elemző meg van írva.

SZIMBÓLUMTÁBLA

A TinyScript nyelvben változókat és függvényeket definiálhatunk, ezek lesznek a szimbólumaink. Mivel egy függvénynek több tulajdonsága van, mint egy változónak (lehetnek paraméterei), illetve máshogy is viselkednek (meg lehet őket hívni), külön kezeljük őket a scope-ban. A **tinyscript.symboltable** package alatt megtalálható a változó, függvény, és scope kezelése. A *FunctionSymbol* osztály *addParameter* függvényére azért van szükség, hogy duplikált paraméter nevek esetén testre szabhatóbb hibaüzenetet tudjunk adni. A *TSType* osztály a típusrendszer egy típusát írja le, erről később lesz szó.

```
public class FunctionSymbol {
    public String name;
    public TSType returnType;
    public final HashMap<String, VariableSymbol> parameters = new HashMap<>();

    public FunctionSymbol(String name, TSType returnType) {
        this.name = name;
        this.returnType = returnType;
    }

    public boolean addParameter(String name, VariableSymbol paramSymbol) {
        if (parameters.containsKey(name)) {
            System.out.println("parameter '" + name + "' is already in scope");
            return true;
        }

        parameters.put(name, paramSymbol);
        return false;
    }

    @Override
    public String toString() { ... }
} }
```

```
public class VariableSymbol {
    public TSType type;
    public String name;

    public VariableSymbol(TSType type, String name) {
        this.type = type;
        this.name = name;
    }

    @Override
    public String toString() { ... }
}
```

A *Scope* osztály kezeli a parent scope-ot (*previous*), valamint támogatja a szokásos lookup és insert műveleteket (*getXY* és *addXY* függvények).

```

public class Scope {
    private final Scope previous;
    private final HashMap<String, VariableSymbol> variables = new HashMap<>();
    private final HashMap<String, FunctionSymbol> functions = new HashMap<>();

    public Scope getPrevious() {
        return previous;
    }

    public Scope(Scope previous) {
        this.previous = previous;
    }

    public VariableSymbol getVariable(String name) { ... }

    public FunctionSymbol getFunction(String name) { ... }

    public boolean addVariable(String name, VariableSymbol symbol) { ... }

    public boolean addFunction(String name, FunctionSymbol symbol) { ... }

    ...
}

```

TÍPUSRENDSZER

A típusrendszer felépítése a **tinyscript.typesystem** package alatt található. Egy típusnak szülőtípusai lehetnek, két típus kompatibilitását ezen reláció mentén tudjuk vizsgálni (*isCompatibleWith* függvény).

```

public class TSType {
    public String name;
    public HashSet<TSType> parents;

    public TSType(String name) {
        this.name = name;
        parents = new HashSet<>();
    }

    public boolean isCompatibleWith(TSType t) {
        return this == t || parents.stream().anyMatch(p -> p.isCompatibleWith(t));
    }
}

```

A *TypeSystem* osztályban található az összes beépített típus, itt adjuk meg a köztük lévő kapcsolatokat is. Az *ERROR* típusnak szülője lesz az összes egyéb beépített típus, a típushibák eszkalációjának elkerülése miatt. Saját típust is felvehetnénk (*createType* függvény, pl. osztályok), de a TinyScript nyelvben erre jelenleg nincs lehetőség. Ekkor kezelniük kellene a *NULL* és *ERROR* típusok szülő kapcsolatait is, például egy új osztály mindkettőnek szülője kell legyen.

```

public class TypeSystem {
    private final HashMap<String, TSType> types;

    public TypeSystem() {
        types = new HashMap<>();
        initializeConstants();
    }
}

```

```

public TSType BOOLEAN;
public TSType INT;
public TSType STRING;
public TSType NULL;
public TSType ERROR;
public TSType VOID;

private void initializeConstants() {
    this.ERROR = new TSType("ErrorType");
    this.STRING = new TSType("string");
    this.INT = new TSType("int");
    this.BOOLEAN = new TSType("bool");
    this.NULL = new TSType("NullType");
    this.VOID = new TSType("void");

    this.NULL.parents.add(this.STRING);
    this.ERROR.parents.add(this.INT);
    this.ERROR.parents.add(this.BOOLEAN);
    this.ERROR.parents.add(this.NULL);
    this.ERROR.parents.add(this.VOID);

    types.put(this.ERROR.name, this.ERROR);
    types.put(this.INT.name, this.INT);
    types.put(this.STRING.name, this.STRING);
    types.put(this.BOOLEAN.name, this.BOOLEAN);
    types.put(this.NULL.name, this.NULL);
    types.put(this.VOID.name, this.VOID);
}

public TSType getType(String name) {
    if (types.containsKey(name))
        return types.get(name);

    return this.ERROR; // type error escalation
}

...
}

```

A *TypeSystemHelper* osztály a típusrendszer és szimbólumtábla (scope) alapján megmondja egy adott kifejezésről, hogy az milyen típusú. A típusrendszert és az aktuális scope-ot a visitor fogja átadni, utóbbi természetesen függ attól, hogy hol járunk a kódban. Bonyolultabb nyelvek esetén további segédosztályok és adatszerkezetek is készíthetők.

```

public class TypeSystemHelper {
    public static TSType getType(TinyScriptParser.ExpressionContext ctx, TypeSystem ts, Scope scope) {
        ...
    }

    public static TSType getType(TinyScriptParser.PrimaryContext ctx, TypeSystem ts, Scope scope) {
        ...
    }

    public static TSType getType(TinyScriptParser.FunctionCallContext ctx, TypeSystem ts, Scope scope) {
        ...
    }

    public static TSType getType(TinyScriptParser.LiteralExpressionContext ctx, TypeSystem ts, Scope scope) {
        ...
    }
}

```

```

    ...
}
}

```

SZINTAXISFA BEJÁRÁSA VISITOR MINTÁVAL

Az ANTLR nyelvtan minden parser szabályából egy saját Context osztály generálódik (pl. *IfStatementContext*), amely típusos API-t biztosít arra, hogy lekérjük a fában a gyerekeit. A visitor minta alapgondolata, hogy szétválasztja az adatszerkezet bejárását attól, hogy az egyes csomópontokban mit kell tennünk. Az ANTLR alapértelmezetten mélységi bejárást (DFS) alkalmaz, amit részben vagy egészben felüldefiniálhatunk a *return* utasítások átszervezésével a *visit* függvényeken belül, de gyakori, hogy erre nincs is szükség.

A **tinyscript** package *TinyScriptSemanticAnalyzer* osztálya felelős a teljes szemantikai elemzés elvégzéséért. Kezelünk egy *scope*-ot (melyet majd dinamikusan változtatunk) és egy típusrendszert (*ts*). Az *exceptionHandler* a hibaüzenetek kezeléséért felelős. A *currentFunction* tagváltozóra azért van szükség, hogy a függvények paramétereinél és a *return* utasításnál tudjuk, hogy melyik függvényben járunk éppen.

```

public class TinyScriptSemanticAnalyzer extends TinyScriptBaseVisitor<Object> {
    private Scope scope;
    private TypeSystem ts;
    private final TinyScriptExceptionHandler exceptionHandler;
    private FunctionSymbol currentFunction = null;

    public TinyScriptSemanticAnalyzer(TinyScriptExceptionHandler exceptionHandler) {
        this.exceptionHandler = exceptionHandler;

        scope = new Scope(null);
        ts = new TypeSystem();
    }
    ...
}

```

A továbbiakban nem fogjuk teljesen végig nézni a szemantikai elemzőt, csak néhány fontosabb részletet emelünk ki. Függvény definíció bejárásánál (*FunctionDefinitionContext*) először betesszük a függvényt az aktuális *scope*-ba és egy új, lokális *scope*-ot nyitunk a függvénynek, hiszen lehetnek lokális tagváltozók, amiket csak a függvényen belül érhetünk el. Ezután a függvény utasításainak bejárása következik (a *super.visitFunctionDefinition*-t kiemeltük, nem térünk vele vissza azonnal), utána pedig visszaállítjuk a *scope*-ot. Figyeljük meg, hogy a *FunctionDefinitionContext* osztályon olyan függvényeket tudjuk elérni, amely parser szabályokat a nyelvtanban definiáltunk!

```

@Override
public Object visitFunctionDefinition(TinyScriptParser.FunctionDefinitionContext ctx) {
    var name = ctx.functionName().getText();
    var returnType = ts.getType(ctx.returnType().getText());
    currentFunction = new FunctionSymbol(name, returnType);
    addFunctionToScope(ctx, currentFunction);

    scope = new Scope(scope);
    System.out.println("Function entry");

    var result = super.visitFunctionDefinition(ctx);

    System.out.println("Function exit");
    System.out.println(scope.toString());
    scope = scope.getPrevious();
}

```

```

        currentFunction = null;

        return result;
    }

```

Return utasítás esetén meg kell néznünk, hogy függvényen belül vagyunk-e, erre is jó a korábban említett *currentFunction* tagváltozó. Ezen kívül a visszatérés típusát is tudjuk ellenőrizni.

```

@Override
public Object visitReturnStatement(TinyScriptParser.ReturnStatementContext ctx) {
    if (currentFunction == null) {
        addException(ctx, "return statements can only be present inside a function");
    }
    else {
        var type = TypeSystemHelper.getType(ctx.expression(), ts, scope);
        if (!type.isCompatibleWith(currentFunction.returnType))
            addException(ctx, "return statement type " + type.name + " is not compatible with function return type " + currentFunction.returnType.name);
    }

    return super.visitReturnStatement(ctx);
}

```

Változó deklaráció során több feladatunk is van: „var” kulcsszó használata esetén ellenőriznünk kell, hogy legyen kezdeti érték; ha van kezdeti érték (de nincs „var” kulcsszó), akkor pedig a változó típusával kompatibilisnek kell lennie a kezdeti értéket leíró kifejezésnek. Ezen kívül a változót be kell szűrnünk az aktuális scope-ba.

```

@Override
public Object visitVariableDeclaration(TinyScriptParser.VariableDeclarationContext ctx) {
    var varName = ctx.varName().getText();
    // var
    if (ctx.VAR() != null) {
        if (ctx.expression() == null)
            addException(ctx, "variables declared using 'var' must have an initial value");
        else {
            var type = TypeSystemHelper.getType(ctx.expression(), ts, scope);
            var symbol = new VariableSymbol(type, varName);
            addVariableToScope(ctx.varName(), symbol);
        }
    }
    // not var
    else {
        var type = ts.getType(ctx.typeName().getText());

        if (ctx.expression() == null) {
            var symbol = new VariableSymbol(type, varName);
            addVariableToScope(ctx.varName(), symbol);
        }
        else {
            var expressionType = TypeSystemHelper.getType(ctx.expression(), ts, scope);
            if (!expressionType.isCompatibleWith(type))
                addException(ctx, "an expression of type '" + expressionType.name + "' is not assignable" + " to variable '" + varName + "' of type " + type.name);
            else {
                var symbol = new VariableSymbol(type, ctx.varName().getText());
                addVariableToScope(ctx.varName(), symbol);
            }
        }
    }
}

```

```

    }
}
System.out.println("Var declaration visited");
System.out.println(scope.toString());
return super.visitVariableDeclaration(ctx);
}

```

A visitor a konzolra kiírja a főbb lépéseket, illetve a talált hibákat is. Érdekes futtatnunk (*TinyScriptRunner* osztály) és végig követnünk, hogy pontosan hogyan működik, valamint az *input.tys* fájl átírásával tesztelnünk az elemzőt.

*A szemantikai elemzés során, amikor szimbólumtáblát (vagy hasonló adatszerkezetet) kell felépítenünk, általában több visitorot készítünk. Az első visitor felépíti a szimbólumtáblát és az építés közben jelzi az esetleges hibákat (pl. ha két globális függvénynek ugyanaz a neve), majd a felépített szimbólumtábla alapján a második (vagy az N-edik) visitor elvégzi a szemantikai elemzés azon ellenőrzéseit, amelyekhez a teljes szimbólumtáblára szükség van (pl. hogy létezik-e az adott függvény). A kiadott kódban mi ezt megspóroltuk, a fa bejárási sorrendjének megváltoztatásával (ld. *visitProgram* függvény) először a függvények kerülnek hozzáadásra a scope-ba. Ha feltesszük, hogy egy függvényen belül nem tudunk más függvényt hívni, akkor ez egy jó megoldás, ellenkező esetben szükségünk lenne egy másik visitorra, ami előbb az előzőek alapján felépíti a szimbólumtáblát.*

III. KÓDGENERÁLÁS

A TinyScript nyelv esetén a transzformáció és optimalizáció kimarad, a szemantikai elemzés után a kódgenerálás következik. A kódgenerálást is a visitor minta alapján végezzük. Egy sablon alapú transpilert fogunk készíteni, ami a TinyScript nyelven leírt kódot Java kóddá alakítja át, a *StringTemplate*² technológia felhasználásával. Mindig pontosan egy Java osztályt generálunk (pl. *TinyScriptOutput*) egy main függvénnyel; a TinyScript nyelven leírt további függvények is ebbe az osztályba fognak generálódni. A generált fájl alapértelmezetten a **tinyscript.generated** package-ben jön létre.

Mivel a TinyScript és a Java nyelv között nagyon nagy a szintaktikai hasonlóság, ezért egyszerű dolgunk lesz a kódgenerálás során. Az utasításokat szinte teljesen át tudjuk venni, néhány kivételtől eltekintve (típusnevek) mindkét nyelvben ugyanaz az utasítások szintaxisa. A kódgenerálás egy része megtalálható a kiinduló projektben, ezt tekintjük át a következőkben. A kódgenerálás további feladatai a házi feladat részét képezik.

*FONTOS: ha hibás Java fájlt generálnánk, lehet, hogy az IntelliJ IDEA (mivel Java projekt van megnyitva) nem engedi futtatni az alkalmazást. Ekkor töröljük ki a generált fájlt és futtassuk újra a *TinyScriptRunner* osztály *main* függvényét.*

Kezdjük a *StringTemplate* használatával: a **tinyscript.codegen.stringtemplate** package alatt található egy úgynevezett group file (*JavaGen.stg*), ami tetszőleges számú sablont (template) tartalmaz. A tényleges kód (visitor) megírása során a sablonokat paraméterekkel látjuk el (pl. *packageName*, *functions*, stb.), valamint tartalmukat más sablonba is beágyazhatjuk. Sablonon belül paramétert a *< >* karakterek használatával tudunk behelyettesíteni. A *java* sablon a generált kód vázát tartalmazza: egy package és osztálydefiníciót, az osztályon belüli függvényeket, valamint egy main függvényt. A **függvények generálása** nincs megoldva, ez a **házi feladat része**. A *statements* sablon nagyon egyszerű, az utasításokat (vagyis tetszőleges stringeket) gyűjti egymás után, újsor karakterrel elválasztva. Ehelyett használhatnánk *StringBuilder*-t is a kódban, de a *StringTemplate* egyik előnye, hogy nincs szükség manuális string összefűzésekre, helyettük inkább felparaméterezett sablonokat használunk.

² <https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>


```

java(packageName, className, functions, mainBody) ::= <<
package <packageName>;

public class <className> {
    <functions; separator="\n">

    public static void main(String[] args) {
        <mainBody>
    }
}
>>

statements(statement) ::= <<
<statement; separator="\n">
>>

```

A **tinyscript.codegen** package *TinyScriptCodeGenerator* osztálya – hasonlóan a szemantikai elemzőhöz – egy visitor, ami a kódgenerálást végzi. Ehhez szükség van a *StringTemplate groupFile*-ra, az ebben található fő sablonra (*javaTemplate*), valamint egy külön tagváltozót készítettünk a *main* függvényen belül található utasításoknak is (*mainStatementsTemplate*), ami a fent említett *statements* sablont használja. A *visit* függvény felüldefiniálásával megadjuk a generálás menetét: példányosítjuk a *statements* sablont, bejárjuk a szintaxisfát, a végén pedig az összegyűjtött utasításokat hozzáadjuk a fő (*java*) sablonhoz, mint paraméter (*mainBody*). Végül a generált kód a *java* sablon által renderelt szöveg lesz.

```

public class TinyScriptCodeGenerator extends TinyScriptBaseVisitor<Object> {
    public String generatedCode = "";
    private final STGroupFile groupFile;
    private final ST javaTemplate;
    private ST mainStatementsTemplate;

    public TinyScriptCodeGenerator(String stGroupPath, String javaTemplate, String packageName, String
className) {
        this.groupFile = new STGroupFile(stGroupPath);
        this.javaTemplate = groupFile.getInstanceOf(javaTemplate);
        this.javaTemplate.add("packageName", packageName);
        this.javaTemplate.add("className", className);
    }

    @Override
    public Object visit(ParseTree tree) {
        mainStatementsTemplate = groupFile.getInstanceOf("statements");

        var result = super.visit(tree);
        javaTemplate.add("mainBody", mainStatementsTemplate);
        generatedCode = javaTemplate.render();
        return result;
    }
    ...
}

```

A szintaxisfa bejárása során minden utasításból készítünk egy stringet (*processStatement*), mely során a két nem Java kompatibilis típusnevet (string és bool) lecseréljük. Egy komplexebb nyelvnél (illetve ahol a célnyelv szintaxisa nem lenne ennyire hasonló) ez persze sokkal bonyolultabb feladat lenne, itt ezt a lépést leegyszerűsítjük. Az *if* és *while* utasításokra külön figyelniünk kell, hogy ne járjuk be őket még egyszer, különben duplikálnánk a generált kódot, mivel ezen utasítások belseje is *statement*-eket tartalmaz. A *getFullSourceText* segédfüggvény arra szolgál, hogy a szintaxisfába nem bekerülő whitespace és újsor karakterek is megjelenjenek a generált kódban.

```
@Override
public Object visitStatement(TinyScriptParser.StatementContext ctx) {
    mainStatementsTemplate.add("statement", processStatement(ctx));

    if (ctx.ifStatement() == null && ctx.whileStatement() == null)
        return super.visitStatement(ctx);
    else return null;
}

private String processStatement(TinyScriptParser.StatementContext ctx) {
    return getFullSourceText(ctx).replace("string", "String").replace("bool", "boolean");
}

private String getFullSourceText(ParserRuleContext context) {
    CharStream cs = context.start.getTokenSource().getInputStream();
    int stopIndex = context.stop != null ? context.stop.getStopIndex() : -1;
    return cs.getText(new Interval(context.start.getStartIndex(), stopIndex));
}
```

*A kódgenerálás során is használhatnánk a szimbólumtáblát vagy típusrendszert, ez bonyolultabb nyelvek esetén tipikusan szükséges is. A mostani feladatnál például a típusnevek konvertálását TinyScript és Java nyelv között (*string* --> *String* és *bool* --> *boolean*) szebben is meg tudnánk csinálni, ha használnánk és kiegészítenénk a *TypeSystem* osztályt, de itt az egyszerűség mellett döntöttünk.*

A kódgenerálás a *TinyScriptRunner* osztályban van felparaméterezve (input és output fájlok, *StringTemplate* sablon) és a *main* függvényt futtatva, a szemantikai elemzés után fut le. A kiinduló projektben található **kódgenerátor hibás**, a függvények utasításait is a *main* függvényen belülre generálja (pl. *return a + b;*), ezt a **házi feladat** során kell majd megoldanunk! A kiinduló projektben található példa bemenetre (*input.tys*) van egy példa kimenet is (*TinyScriptOutputExample.java*).