**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Ónodi-Kiss Viktor Ákos

12. osztályos tanuló

# PROGRAMOZÁSI NYELV KÉSZÍTÉSE JÁTÉKOS OKTATÁS CÉLJÁRA

Konzulens

Dr. Somogyi Ferenc Attila

BUDAPEST, 2023

# Tartalomjegyzék

Összefoglaló ........ 4

Abstract ........ 5

1. Bevezetés ........ 6

1.1 Témaválasztás indoklása ........ 6

1.2 Megoldandó feladat leírása ........ 7

1.3 A dolgozat struktúrája ........ 8

2. Irodalom kutatás ........ 9

2.1 Célközönség ........ 9

2.2 A Hedycode ........ 9

2.3 CodeCombat ........ 10

2.4 Code.org ........ 11

2.5 Kérdőív ........ 12

3. Elméleti háttér és felhasznált technológiák ........ 13

3.1 Programozási nyelvek átalakítása gépi kóddá ........ 13

3.1.1 Fordítóprogramok ........ 13

3.1.1.1 Lexikai elemzés ........ 14

3.1.1.2 Szintaktikai elemzés ........ 14

3.1.1.3 Szemantikai elemzés ........ 15

3.1.1.4 Optimalizáció ........ 16

3.1.1.5 Kódgenerálás ........ 16

3.1.2 Interpreterek ........ 17

3.2 Felhasznált technológiák ........ 18

3.2.1 ANTLR ........ 18

3.2.2 AvalonEdit ........ 19

3.2.3 Nakov.TurtleGraphics ........................................................................................ 20
4. Megvalósítás ........................................................................................ 21
4.1 A nyelv felépítése ........................................................................................ 21
4.2 Visitor ........................................................................................ 24
4.2.1 Ellenőrző ........................................................................................ 24
4.2.1.1 Hibakezelés ........................................................................................ 25
4.2.2 Interpreter ........................................................................................ 25
4.3 Grafikus keretrendszer ........................................................................................ 26
5. Szemléltető példa ........................................................................................ 28
6. Összefoglalás ........................................................................................ 30
6.1 Továbbfejlesztési lehetőségek ........................................................................................ 31
6.2 Köszönetnyilvánítás ........................................................................................ 32
Irodalomjegyzék ........................................................................................ 33

# Összefoglaló

Napjainkban a programozás egy egyszerű és jól kereső álláslehetőségnek tűnik sok fiatalnak. Oktatóként azonban gyakran tapasztalom azt, hogy akik a programozási ismeretek elsajátítására adják a fejüket, azok az esetek nagyobb részében elakadnak, majd akár fel is adják az alapoknál, mivel nehézséget okoz nekik annak elsajátítása.

Ennek a problémának az orvoslására készítettem egy saját programozási nyelvet, mely elő hivatott segíteni a kezdő programozók tanulmányait, továbbá a régebbi programozás oktatási gyakorlatokból leporolja a sokak által ismert teknőst is.

Dolgozatomban először felmérem a kezdő programozók számára okozott legnagyobb nehézségeket, melyeket a nyelv kidolgozása során figyelembe vettem. Ezenkívül ismertetem a klasszikus fordítóprogramok működését, bemutatom az általam kidolgozott nyelvet, illetve kifejtem, hogy a nyelv miben segíti az oktatási problémák megoldását.

# Abstract

Today, programming seems to many young people as an easy and well-paid profession. However, as an educator, I often find that those who put their minds to learning programming skills are more often than not get stuck and even give up at the basics because they find it difficult to master.

To remedy this problem, I have created a programming language of my own, which is intended to help beginner programmers study, and also attempt to dust off the turtle that many people are familiar with from older programming education practices.

In my thesis, I first assess the main difficulties faced by novice programmers, which I have considered when developing the language. I will also describe how classical compilers work, present the language I have developed, and explain how the language helps to solve educational problems.

# 1. Bevezetés

## 1.1 Témaválasztás indoklása

Egészen fiatal korom óta érdekel a programozás oktatása, már általános iskolában én voltam az informatika tanár jobb keze az iskolák utáni informatika szakkörön, ahol én segítettem a kisebbeknek megérteni a programozást. Ezt követően, középiskolás éveim alatt kezdtem rendesen programozást tanulni és bele is szerettem a programozás mögött rejlő elméletbe, mindig szemem előtt tartva a mondást, miképp „Ha nem tudod elmagyarázni egy 6 évesnek, akkor te sem érted igazán”.

A programozás oktatás módszere nagyban eltér a többi tantárgy/szak tanításától és bár mindegyiknek megvannak a saját különlegességei, mégis a programozás az egyetlen amelyiket teljes mértékben digitalizálni lehet. Ezt a digitalizációt, bár erőltetetten, de mégis előre segítette a COVID járvány, mely alatt, az online oktatásnak köszönhetően, óriási mértékben előre lépett az digitális oktatás. Több „offline” bootcamp digitálissá tette a tanfolyamait és online tutorialok előnybe kerültek a papír alapú könyvekkel szemben, ezzel is megerősítve a programozás helyét a világban, mint valamit, aminek a tanulásához a potenciális tanuló inkább „olvasson utána a neten”.

Azonban az online fellelhető módszerek gyakran ijesztőek lehetnek egy fiatal diáknak, akinek nincs pénze a különböző drágábbnál drágább bootcampekre, illetve akit könnyen megrémíthetnek a nehezebb koncepciók.  Ezért választottam a játékos programozást, mint dolgozatom témáját, mert szeretnék próbát tenni egy olyan nyelv/környezet készítésében ami segít a fiatal programozóknak megérteni a kódolás alapjait, játékos formában.

## 1.2 Megoldandó feladat leírása

A dolgozat témájának kiválasztása után kitűztem magam elé célnak, hogy a projektemnek mindenképpen tartalmaznia kell egy szoftveres prototípust, hogy demonstráljam az elméleti munka eredményét.

A dolgozatom mögött rejlő projektet négy fő részre tudom bontani:

1. A programozás alapjainak elméleti oktatására vonatkozó megfigyelések, tapasztalatok
2. A programozási nyelvek futtatását végző fordítóprogramok (compilerek) és értelmezők (interpreterek) működésének megismerése
3. Az elkészített programozási nyelv [1]
4. A nyelvhez készített mini-szoftverfejlesztői környezet (IDE)

A nyelvet Boascriptnek neveztem el, elsődlegesen azért, mert olyan nyelvnek szántam, mely magában foglalja a C# programozási nyelv OOP (objektum orientált programozás) paradigmáját és a nyelv bizonyos szintaktikai felépítését, illetve a Python átláthatóságát. Továbbá a nyelv nevében megtalálható a „script” kifejezés is, egyfajta tiszteletadásként a JavaScript előtt, melynek szintaktikájából ugyancsak merítettem ötletet a nyelv megírásakor. Így született meg a Boascript, vagy „Boa”, mely a „.boa” kiterjesztésű forráskódfájlokról ismerhető meg.

Ehhez tevődik hozzá, egyfajta kiegészítőként, a szoftverfejlesztői környezet, mely lehetővé teszi a teknős könyvtár használatát, illetve tartalmazza a beépített dokumentációt és feladatokat, melyekkel a felhasználó el tudja mélyíteni a saját programozási ismereteit.

## 1.3 A dolgozat struktúrája

A dolgozat a következő fejezetekből épül fel:

A 2. fejezetben bemutatom, hogy milyen technológiákat ismertem meg a dolgozat megírását megelőző kutatás közben, illetve azt is, hogy milyen tanulságokat vontam le belőlük a dolgozat elkészítéséhez.

A 3. fejezet lényege a felhasznált technológiák ismertetése és a dolgozat megírása mögötti elmélet leírása. Ebben a fejezetben mutatom be a fordítóprogramok és az értelmezők működését is.

A 4. fejezetben ismertetem a nyelv különlegességeit és azt is, hogy milyen céllal helyeztem el őket a nyelvben. Ezen kívül itt beszélek röviden az elkészített mini-IDE-ről is.

Az 5. fejezetben egy rövid szemléltető pédán keresztül mutatom be a nyelv és mini-IDE működését.

Végül a dolgozat 6. fejezetében összefoglalom a dolgozat eredményeit, a személyes fejlődést melyet az írása közben értem el, illetve bemutatom a lehetséges továbbfejlesztési irányokat is.

# 2. Irodalom kutatás

## 2.1 Célközönség

A nyelv elkészítésekor elsődlegesen, a saját programozás oktatási tapasztalatomból adódóan, az általános iskola felső évfolyamaiba (5-8. osztály) és a középiskolákba járó diákokra fordítottam a fókuszt. A dolgozaton való munka elkezdésekor készítettem egy kérdőívet, melyet a lehető legtöbb embernek (programozóknak és nem programozóknak egyaránt) megpróbáltam eljuttatni és a kérdőív eredményei megmutatták, hogy míg a tapasztaltabb programozóknak is vannak lyukak tudásukban, ez gyakran betudható a Dunning-Kruger effektnek [2], ezzel szembe a fiatal programozók elveszve érzik magukat a programozás végtelennek tűnő világában.

Ezért is döntöttem úgy, hogy elsődlegesen az utóbbi csoportra fektetem a fókuszomat, ugyanis a kérdőív alapján fel tudtam használni az idősebbek óva intéseit a fejlesztéskor.

## 2.2 A Hedycode

A Hedy programozási nyelvre [3] a dolgozatra való felkészülésként végzett kutatómunka közben találtam rá. A Hedy lényege egy olyan programozási nyelv megalkotása volt, mely egyszerűbbé teszi a programozás oktatást. Ezt a következő módon éri el:

- **A Hedy graduálisan épül fel**

  Ez azt jelenti, hogy a nyelv megtanulása szintről-szintre történik. Először a tanuló megismerkedik a kimeneti parancsokkal (print()) és csak ezt követően van belevezetve a kicsivel komplikáltabb algoritmikai elemekbe, mint például a vezérlési szerkezetek, ciklusok.

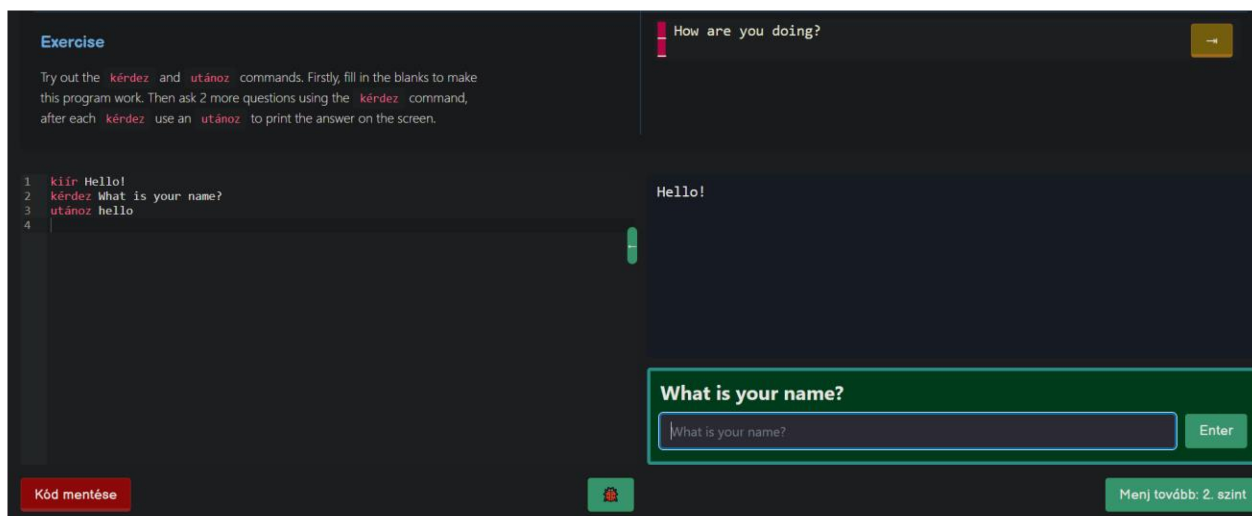
- **A forráskód nyelve szabadon változtatható**

  A Hedyhez biztosított egy szövegszerkesztő, mely a nyelv hivatalos weboldalába [3] van beépítve. Ez a szövegszerkesztő lehetővé teszi a forráskód nyelvének megváltoztatását egy olyanra mely a tanulóhoz közelebb áll. Ezt a nyelvi szótárat lelkes önkéntesek tartják fenn és már közel 50 nyelvet tartalmaz.

- **A programozási élmény teljesen oktatásra szabható**

  A tanároknak lehetőséget biztosít a weboldal a beépített óra sémák használatára, de akár saját tanórákat is összerakhatnak, melyet be tudnak tölteni a hedy felületébe, ezzel szinte teljes lehetőséget adva az oktatás menetének irányítására

Ezzel együtt előnye még a Hedycodenak az, hogy a Python nyelvre alapul a szintaktikája, mely egy egyszerűbb felépítést biztosít a magas szinten való elhelyezkedése miatt. Továbbá, az eddigi tapasztalataim alapján a Python nyelv ismerete egy kedvező alapot tud adni a programozás tanulás megkezdésére, azonban azoknak akik később kezdik el tanulni, már komolyabb nehézséget tud okozni az elsajátítása az egyedi szintaktikai felépítése miatt.




**1. ábra: A Hedycode 1. szintje**

## 2.3 CodeCombat

Abban az esetben, ha meg akarunk tekinteni egy olyan példát melynek a használatáért fizetni kell, akkor meg tudjuk tekinteni a CodeCombat-ot [4], mely lehetőséget ad a tanulóknak (a megfelelő összeg kifizetése után) a graduális programozás tanulást, Pythonban, JavaScriptben és C++-ban. A Hedyvel ellentétben a CodeCombat végeredménye nem egy egyszerű konzolos be- és kimenet, hanem egy teljes játék, melyet a tanulók a böngészőben tudnak futtatni. Ezzel együtt a CodeCombat tartalmaz egy beépített súgót és osztálytermi támogatást is.

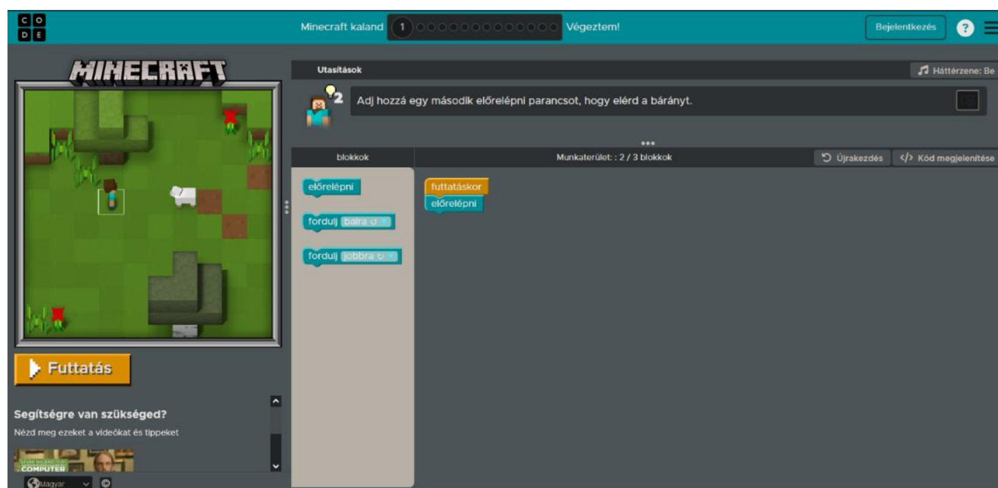
## 2.4 Code.org

A code.org [5] egy olyan weboldal, melynek nevét több fiatal diák ismerheti, ha legalább általános iskolában volt egy kicsivel programozás oktatásban érdekeltebb informatika tanáruk, aki az „hour of code” keretei között bemutatta nekik a weboldalt. Beismerem, jómagam is így ismertem meg.

A code.org lényege egyszerűbb algoritmusok összerakása egy blokk alapú programozási nyelvben egyszerűbb feladatok alapján. Kiemelendő előnyei többek közt, hogy a tanuló a saját nyelvén használhatja a weboldalt (a blokkok nyelve is azon a nyelven jelenik meg), illetve, hogy a feladatok stilizálva vannak közismert szórakoztató médiumok alapján, mint például a Minecraft videójáték és a „Jégvarázs” c. film.

Oktatási módszertant illetően, ez a weboldal is graduálisan tanítja a programozást, szintről-szintre nehezednek a feladatok, közben pedig a weboldal hasznos tanácsokkal látja el a tanulót.

Ezzel együtt a code.org lehetőséget nyújt a tanároknak egy tantervi katalógus segítségével, melyből szabadon választhatnak átláthatóan felépített tantervek közül.



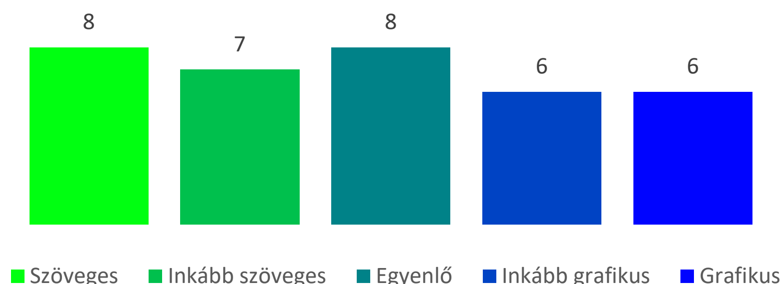

**2. ábra: A Code.org Kódolás órája, "Minecraft kaland" gyakorlatának első szintje**

## 2.5 Kérdőív

Az irodalomkutatás részeként készítettem egy kérdőívet [6] is, melyben többek közt feltettem olyan kérdéseket, mely alapján megtudtam a válaszoló programozási tapasztalatait, a programozás tanulási múltját, kikértem a véleményét példa kódokról és megkérdeztem, hogy mit változtatnának, ha elölről kezdhetnék a tanulást.

A kérdőívet 37 ember töltötte ki a kiértékelés idejéig és a válaszok alapján a kitöltők között voltak középiskolás diákok, egyetemi hallgatók és a szakmában aktívan dolgozó informatikusok is. Ennek köszönhetően a kérdőívre érkezett válaszok egy kiterjedt spektrumot térképeztek fel, amelyben a legkevesebb programozással töltött év 0-volt, a legtöbb pedig 40.

A „Melyiket volt egyszerűbb megérteni [grafikus vagy szöveges programozási nyelveket] (ha már találkoztál mind a kettővel)?” kérdésre a következő válaszok érkeztek:




**3. ábra: A kérdésre adott válaszok eloszlása**

Illetve a „Mi okozta a legnagyobb nehézséget a programozás logikájának megértésben?” kérdésre érkeztek fontos válaszok, melyeket később figyelembe vettem. A kérdésre 19-en válaszoltak az „osztályok” válasszal, 11-en a „függvények/metódusok”-kal, 6-an a „ciklusok”-kal és 6-an a „szintaxis/szemantika”-kal.

Későbbi kérdésekre adott válaszokból kiderült, hogy több embernek is problémát okozott az OOP ismeretek elsajátítása, a motiváció hiánya, a programozás átláthatatlansága és a hiányzó magyarázat egyes új tananyagok tanulásakor.

A kérdőívre adott válaszokból levont következtetéseket felhasználtam a dolgozat későbbi megírásához és a Boascript nyelv működésének és szintaktikájának felépítésekor.

# 3. Elméleti háttér és felhasznált technológiák

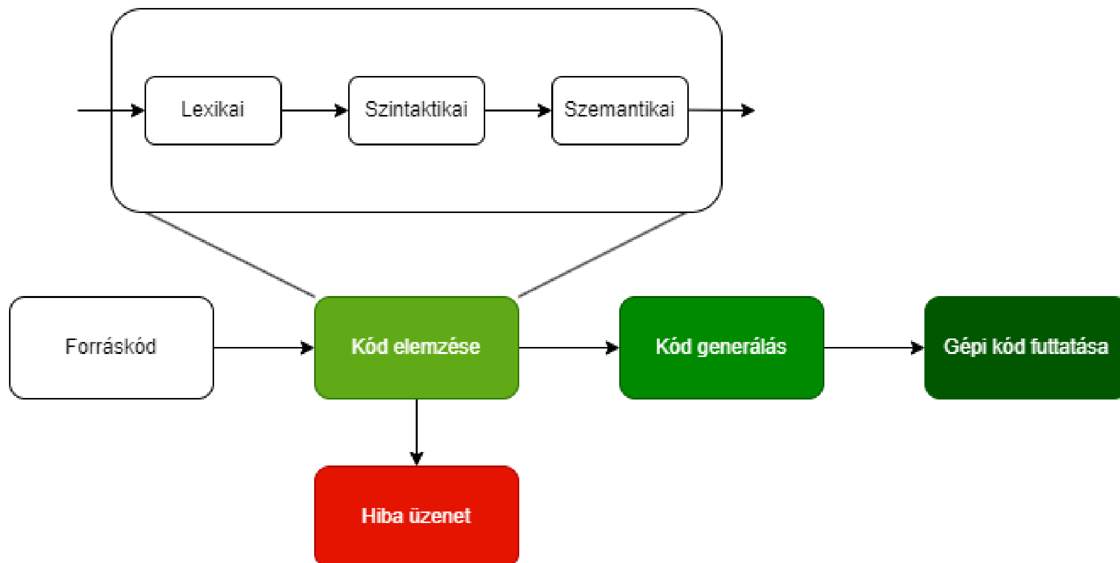
## 3.1 Programozási nyelvek átalakítása gépi kóddá

Szinte minden magas szintű programozási nyelv két kategóriába választható szét a futtatásuk módja szerint, a lefordított (compile-olt) nyelvek, mint például a C#, vagy a Java és az értelmezett (interpretált) nyelvek, melyre a talán legközismertebb példa a Python és a Perl. A fordító és az értelmező programok között a legalapvetőbb különbség, hogy a fordítóprogram egészben olvassa be a forráskódot, majd úgy hajtja végre, ezzel szemben az értelmező sorról-sorra olvassa be a kódot és úgy is hajtja végre az abban szereplő parancsokat [7].

Ezek mellett harmadik kategóriaként megemlíteném (bár a dolgozatom nem foglalkozik vele komolyabban) az összeépítőt, vagy angolul az assembler-t, mely nem keverendő össze az assembly programozási nyelvvel, bár a nevük közötti hasonlóság nem alaptalan, mégis az assembler-t általában arra használjuk, hogy az assembly kódot 0-ák és 1-esek kettes számrendszerben értelmezendő sorozatává, gépi nyelvvé, alakítsuk.

### 3.1.1 Fordítóprogramok

A compiler-ek olyan programok, melyek a magas szintű programozási nyelveket gépi kóddá alakítják. Ezt olyan módon teszik meg, hogy az egész forráskódot beolvassák, értelmezik és ha nem tartalmaz hibát, végrehajtják. Ez a fordítás tipikusan a következő lépésekből áll:




**4. ábra: A fordítóprogramok működése**

#### 3.1.1.1 Lexikai elemzés

Ekkor történik a bemeneti karakterfolyam feldolgozása és a karakterek csoportosítása a fordító program kódjában meghatározott lexer szabályok szerint. Így értelmes egységek, úgynevezett lexémák, jönnek létre. Ilyen lexéma például a változó deklarációban a „var” kifejezés, melyet az azt alkotó karakterek sorozatából ismer fel. Ezt követően, a lexikai elemzés során születnek meg a tokenek is, melyeket a megfelelő lexémák sorozatából alkotunk meg. Ilyen token lehet a, hogy maradjunk az előző példánál, a „var” kifejezésből létrejött VAR token. A lexikai elemzést végző kód innentől már sejti, hogy mire kell számítania mint következő lexéma/token és ha az nem teljesül, akkor hibakódot dob vissza.

Ezt az elemzést általában a dedikált lexer komponens végzi el és a lexikai elemzés kimenete a tokenek sorozata.

Bemenet

var a : int = 12

Kimenet

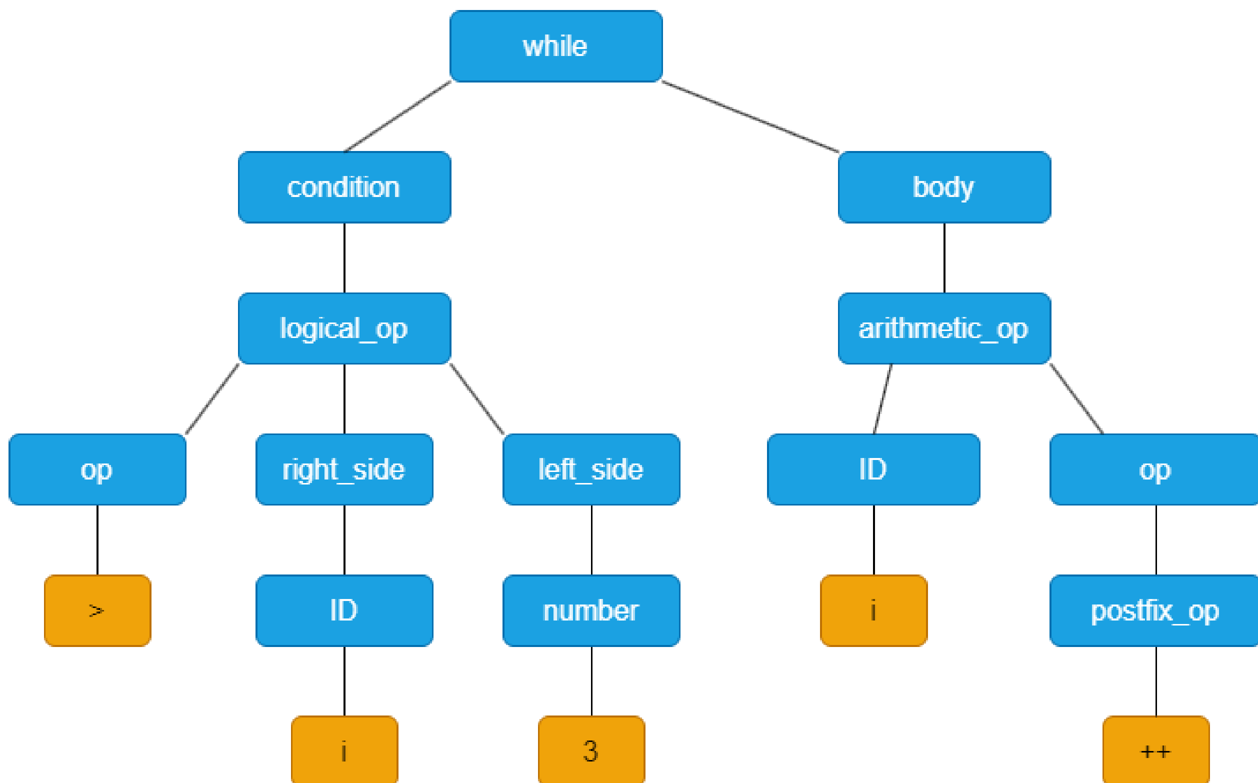
| "var" | "a" | ":" | "int" | "=" | "12" |
|---|---|---|---|---|---|
| Változó deklaráció | ID | típus jelölő | típus | Egyenlő | INT |

**5. ábra: A változó deklaráció lexikai elemzéséből keletkező tokenek**

#### 3.1.1.2 Szintaktikai elemzés

A szintaktikai elemzés célja, hogy a lexikai elemzés során előállt tokenekből egy struktúrát, általában valamilyen szintaxisfát állítson elő. Ilyen token sorozat lehet például az elöl tesztelő while ciklus, ahol egy „while” tokent követ a feltétel, mely után a ciklus „teste” található. Mind ezt pedig egy záró token (általában egy pontosvessző, de lehet sortörés is) követi.

A szintaktikai elemzést az úgynevezett „parser” komponens végzi, a kimenete egy levezetési fa (parse tree) (*ábra 6*). Ebben a fa struktúrában van leképezve az eredeti bemeneti szöveg




**6. ábra: A while ciklus szintaktikai elemzéséből keletkező fa struktúra**

### 3.1.1.3 Szemantikai elemzés

Itt történik a nyelv logikájának vizsgálata. Ilyen logika például az, hogy egy feltételben csak is azonos típusú értékek lehetnek mind a kettő oldalon (pl. az „(5 > 'macska')" feltétel nem értelmezhető), vagy például az is, hogy a felhasználó által beolvasni próbált fájl létezik-e.

Ebben a lépésben történik továbbá a szimbólumtábla (*ábra 7*) elkészítése és karbantartása is, mely magában tárolja minden változó nevét, típusát és értékét, hogy később a kódban hivatkozni lehessen rá, illetve, hogy később meg lehessen változtatni az értékét. Abban az esetben, ha a szimbólumtábla nem tartalmazza a hivatkozott változót, vagy annak hibás a típusa/értéke, akkor ez is jelzésre kerül hibaüzenetként. A szimbólumtáblák megalkotásánál fontos a változó scope-ja is, ugyanis például (bár ez nyelvtől függő) a függvényeken belül létrehozott változók csak is a függvényen belül érhetőek el és változtathatóak.

| ID | Típus | Érték |
|---|---|---|
| Main scope | | |
| a | int | 12 |
| b | string | "hello" |
| gyokVonas | function | √x |
| gyokVonas scope | | |
| x | float | {dynamic} |

**7. ábra: Példa egy szimbólumtábla felépítésére**

#### 3.1.1.4 Optimalizáció

Ekkor történik a forráskód átalakítása olyan formába, hogy később a futtatáskor gyorsabban fusson le, ugyanazzal az elvárt eredménnyel. Ilyen optimalizáció lehet gép független optimalizáció, mely a kódot alakítja át a gyorsabb futásért, és a gép függő optimalizáció, mely átalakítja a kódot, hogy a futtató architektúrával megnövelje a kompatibilitását.

#### 3.1.1.5 Kódgenerálás

Ez az utolsó feladata a fordítóprogramnak, mely csak akkor futhat le, ha a kód fordításának eddigi lépései probléma nélkül lefutottak. Ebben a lépésben futtatja le a legenerált gépi kód béli futtatható fájlt a program, lineárisan, ekkor kerülnek végrehajtásra a kódban szereplő parancsok, ekkor értékelődnek ki a matematikai/logikai műveletek, ekkor megy végig a ciklusokon, ekkor íródnak ki a konzolra a kimenetetek és ekkor mozog előre a teknős.

A fordítóprogramok előnye, hogy (általában) gyorsabbak a többi fordítási módnál, illetve, hogy egy futtatható, platformtól függő kiterjesztésű (Windows esetében például .exe, Androidon .apk) fájlt hoznak létre.

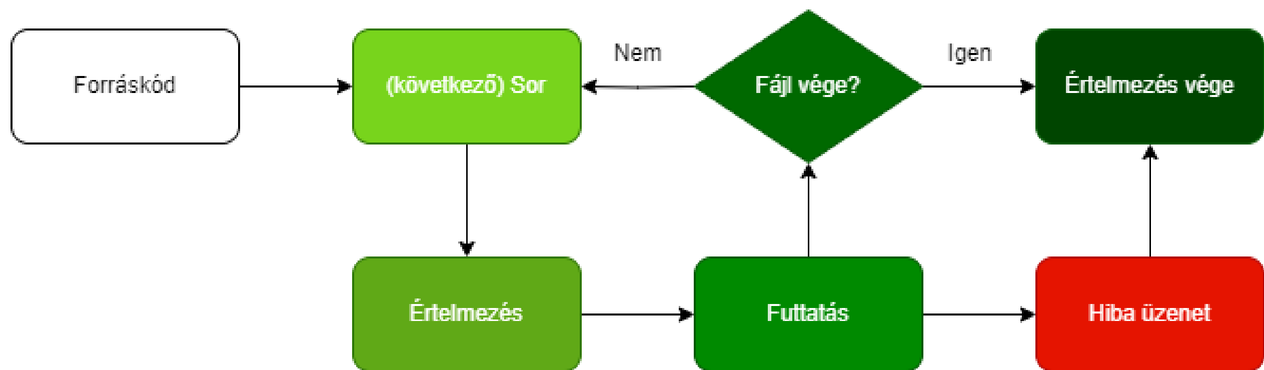
Ezért cserébe viszont minden változtatásnál újra le kell fordítani a kódot, mely sok időbe telhet, továbbá több memóriába kerül a fordítás, ezzel függővé téve a sebességét a gép hardver konfigurációjától.

### 3.1.2 Interpreterek

A fordítóprogramokkal szemben, az értelmezők a kódot sorról-sorra olvassák be, majd hajtják végre. Minden egyes sorban lefut egy ellenőrzés, mely a sor futtathatóságát figyeli meg, ha egy sor nem futtatható, a futtatást abbahagyja.

Ennek előnye a könnyebb debuggolás (hiba mentesítés), mivel sorról sorra futtatva egy rövid kiiratással megtalálhatjuk, hogy hol marad abba a kódunk futása, illetve könnyebben lehet több különböző platform között használni.

A hátránya pedig az, hogy a kód értelmezéséhez a cél eszközön letöltve kell lennie az értelmezőnek és nagyobb programok futtatásánál érezhetően lassabban fut le.




**8. ábra: Az interpreterek működése**

## 3.2 Felhasznált technológiák

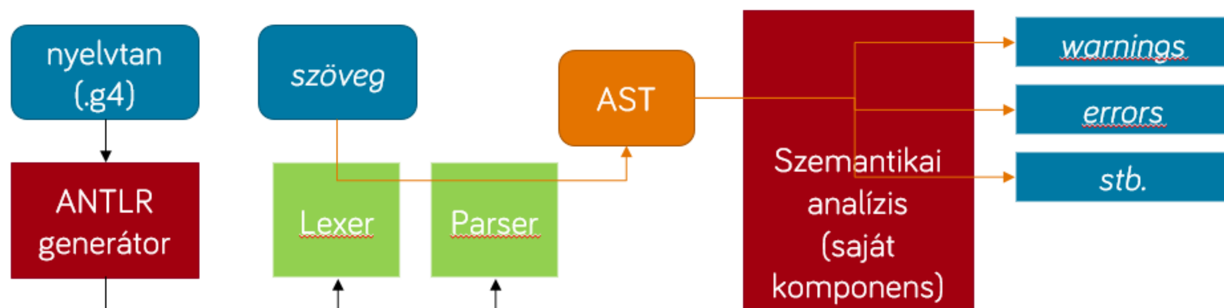
A programozási nyelv fordítóprogramjának megírásához a C# programozási nyelvet használtam fel, mely egy tökéletes középutat adott a magasabb szintű nyelvek, mint a Python, átláthatósága és az alacsonyabb nyelvek kezelhetősége között. Illetve a C# nyelv használata lehetővé tette számomra, hogy a Visual Studio 2022-be beépített WPF UI [8] keretrendszert felhasználjam a szoftverfejlesztői környezet megvalósítására. Az is segített a nyelv választásában, hogy a C# nyelvvel kompatibilis az ANTLR értelmező könyvtára.

### 3.2.1 ANTLR

Az ANTLR [9] egy olyan eszköz, melyről a projekt elkészítése előtt nem sok tudomásom volt, de a munka (és az ahhoz tartozó kutatás) elkezdése óta szert tettem a megfelelő ismeretekre a projekt megvalósításához.

Az ANTLR segítségével a szöveget fel tudjuk dolgozni olyan módon, hogy alapvető elemeket, úgynevezett „token”-eket, ismerjünk fel a saját nyelvtani szabályaink alapján Az ANTLR automatizálja a lexikai és szintaktikai elemzés folyamatát, a bemeneti szövegből képes előállítani (az adott célnyelvre) egy szintaxisfát. A fát bejárva aztán elvégezhető a szemantikai elemzés és a későbbi lépések is.

Az ANTLR kétféle nyelvtani szabály leírását támogatja. A *lexer szabályok* lényege, hogy egy reguláris kifejezés alapján ismer fel tokeneket, például a Boascriptben a szöveg típust úgy ismeri fel, hogy minden olyan karakter, ami két idézőjel között található, egy szövegként, azaz stringként van értelmezve. Ugyanígy a lexer szabályok között van meghatározva, hogy mi jelöli a parancsok végét (a Boascriptben a sortörés karakter, „\n”) és mi az amit figyelmen kívül kell, hogy hagyjon az értelmező.




**9. ábra: Az ANTLR API működése, komponensek szerint**

A *parser szabályok* tartalmazhatnak más (akár lexer, akár parser) szabályokat, de akár önálló szöveg részleteket is. Ezek írják le a nyelv struktúráját, ezek alapján fog szintaxisfa generálódni. Erre példa a for ciklus esetében a „for” kifejezés, viszont a for ciklus feltétele még egy parser szabály magában.

Ezen kívül az ANTLR támogatást nyújt a szemantikai elemzés és későbbi lépések elvégzésére. Ennek alapján a visitor minta [10] képezi, a generált visitor osztályok segítségével egyszerűen elvégezhető a szemantikai elemzés, kódgenerálás, vagy interpretálás is.

### 3.2.2 AvalonEdit

A szoftverfejlesztői környezet elkészítésben nagy segítségemre volt az AvalonEdit [11], mely egy szövegszerkesztő komponens a WPF-hez. Ennek az eszköznek a felhasználásával sikeresen implementáltam egy Boascript kompatibilis szövegszerkesztőt.

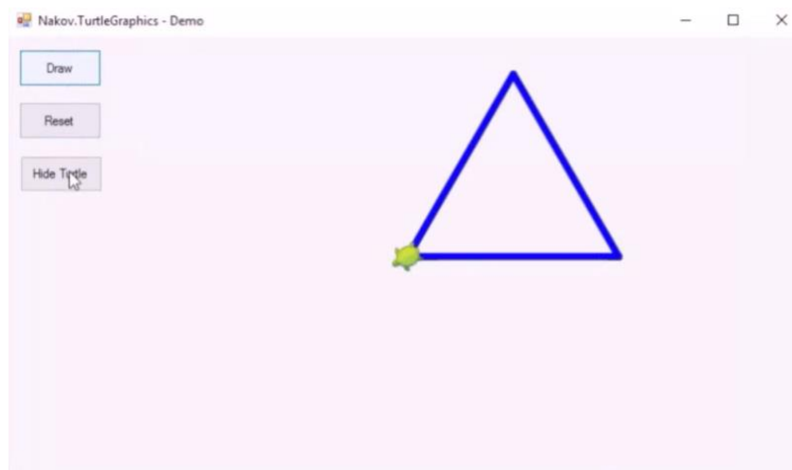
Ebben nagy segítségemre volt az AvalonEdit beépített szintaxis kiemelő funkciója, melyhez elkészítettem egy új szintaxis színezési szabály leíró (xshd) fájlt, abba összegyűjtve a, korábban ANTLR segítségével leírt, lexer szabályokat.

Ez is hozzásegít a Boascript nyelv átláthatóságához és könnyen tanulhatóságához, ugyanis a megfelelő kulcsszavak beírásának helyességét a szövegszerkesztő az annak megfelelő színre való színezésével igazolja vissza.

### 3.2.3 Nakov.TurtleGraphics

A szoftverfejlesztői környezet utolsó, de mégis legfontosabb felhasznált könyvtára a Nakov féle teknős könyvtár [12], mely lehetővé tette a teknős kódok értelmezését majd futtatását C#-ban. A teknős ablak akkor jelenik meg futtatás közben, amikor a megfelelő utasítások meghívásra kerülnek. Ezt követően a program futásának végéig nyitva marad. Az ilyenkor felugró ablak nem tartalmaz semmilyen mértékegységet, ami lehetővé tenné a teknős által megtett út hosszának mérését, az ablak átméretezhető, így a felhasználó szabadon változtathatja a megjelenítés méretét.

A könyvtárból majdnem minden parancs beépítésre került a Boascript-ben, a felhasználó szabadon változtathatja a teknős irányát, léphet vele, lelassíthatja, felgyorsíthatja, illetve még el is tüntetheti. A teknős a logo programozási nyelven alapult. [13]




**10. ábra: Nakov.TurtleGraphics működési demó**

# 4. Megvalósítás

## 4.1 A nyelv felépítése

A Boascript nyelvtanának elkészítésekor elsődlegesen a Python programozási nyelv szintaktikáját vettem alapul, ugyanis az 2. fejezetben említett kutatáskor azt fedeztem fel, hogy a Python egy könnyen megérthető és átlátható nyelvi alapot biztosít a tanulóknak. Továbbá, mint ahogy azt már korábban említettem, a Pythonhoz hasonló nyelvek elsajátítását érdemes minél hamarabb elkezdeni a programozás tanulási út elején lehetőleg. Ezzel együtt a nyelv a „camelCase” nevezési konvenciót követi.

A másik fontos szempont, amit a szemem előtt tartottam, a programozási nyelvekben gyakran előforduló elemek („var” kifejezés) beépítése. A nyelv dinamikusan ismeri fel a változó típusát és ezért nem tartalmaz castingolásra/parsingolásra használható függvényeket.

```
1 printLine("Hello world!")
2
3 print("Hello ")
4 print("újra!")
5
6 inputLine("Adj meg egy számot: ")
7 input("Most egy sorba: ")
```

**11. ábra: A be- és kimenet szintaxisa**

A be- és kimeneti parancsok elkészítésekor külön figyelmet fektettem a sortörésekre, így a Boascript külön parancsot tartalmaz arra, hogy az adott sorban vagy a következőben tartózkodjon a kurzor.

```
BOOLEAN : ('true'|'false');
NULL : ('N'|'n')'ull';
CHAR : '\'' . '\'';
STRING : ('"'|'\'') .*? ('"'|'\'');
FLOAT : ([0-9]*|) ('.'+) [0-9]+;
INT : [0-9]+;
```

**12. ábra: A primitív típusok értékeinek leírására szolgáló lexer szabályok**

A Boascript ugyan tartalmaz dinamikus típusokat, de a felhasználó előre meg is határozhatja a változó típusát, ha szeretné. Ebben az esetben a program hibát ír ki, ha a megadott érték nem kompatibilis típusú. A listák esetében azonban kötelező meghatározni a lista típusát, ugyanis a szimbólum táblába csak az alapján lehet elmenteni az értékét.

```
1 var szoveg = "alma"
2
3 var szam : int = 12
4
5 var lista : list : float = [3.14, 1.5]
```

**13. ábra: A változók deklarációjára példa**

A nyelv három ciklus típust tartalmaz: az első a közismert *for* ciklus, mely majdnem minden nyelvben azonosan néz ki, a második a *foreach* ciklus, melynek a felépítését a C#-ban

```
1  for(var i : int = 0; i < 5; i++){
2      printLine(i)
3  }
4
5  foreach(var elem in lista){
6      printLine(item)
7  }
8
9  loop(4){
10     print("\__/")
11 }
```

**14. ábra: A ciklusok szintaktikai felépítése**

található implementációból vettem át. A harmadik egy különlegesség, a *„loop”* ciklus pontosan annyi alkalommal fut le, mint amennyi a zárójelben található. Ez felgyorsítja az oktatás folyamatát, ugyanis a tanár már az oktatás elején felhasználhatja a kódjában, anélkül, hogy a „sima” for ciklushoz szükséges magyarázatot el kelljen mondania.

```
forStatement: FOR BRACKETS variableDeclaration ';' ((expression';')|(BOOLEAN';'))  assignStatement BRACKETE inBlock;
loopStatement: LOOP BRACKETS INT BRACKETE inBlock;
foreachStatement: FOREACH BRACKETS ID (IN|':') varName BRACKETE inBlock;

whileStatement : WHILE BRACKETS (expression|BOOLEAN|varName) BRACKETE inBlock;
dowhileStatement : DO inBlock WHILE BRACKETS (expression|BOOLEAN|varName) BRACKETE;
```

**15. ábra: A ciklusokat leíró parser szabályok**

Ha egy részletet ki akarok emelni a Boascriptből, mely véleményem szerint eléggé érdekes ahhoz, hogy a nyelvet egyedivé tegye, akkor ki emelném a matematikai és a logikai operátorok működését. A Boascriptben ugyanis a felhasználónak lehetősége van arra, hogy a kívánt operátort úgy írja le, ahogy azt természetes nyelvvel kifejezné (*ábra 16*).

Így lehetséges az, hogy a „[változó név] is incremented by 1" ugyanazt a parancsot jelenti, mint a „[változó név]++". Személyes tapasztalatból fakadóan akartam ezt a nyelv szintaktikájába bele építeni, tekintve, hogy oktatáskor mindig ilyen módon is hivatkozunk az operátorokra és íráskor megzavaró tud lenni a tanulónak, ha mást kell írnia, mint amit hall.

```
1  printLine(szam == 12)
2  printLine(szam is equal to 12)
3
4  //szam++
5  szam is incremented by 1
6
7  //szam = szam - 2
8  szam = szam minus 2
9
10 //szam^2
11 szam = szam to the power of 2
```

**16. ábra: Matematikai és logikai operátorok a Boascriptben**

A nyelv tartalmaz függvényeket is, melynek a szintaktikai leírása a 17. ábrán látható módon néznek (*ábra 17.*).

```
1 function negyzet(x){
2     return x*x
3 }
```

**17. ábra: A függvények szintaktikai felépítése**

Továbbá a nyelv tartalmaz osztályokat is, melyek teljes tartalma egy önálló scope-al rendelkezik (*ábra 18*).

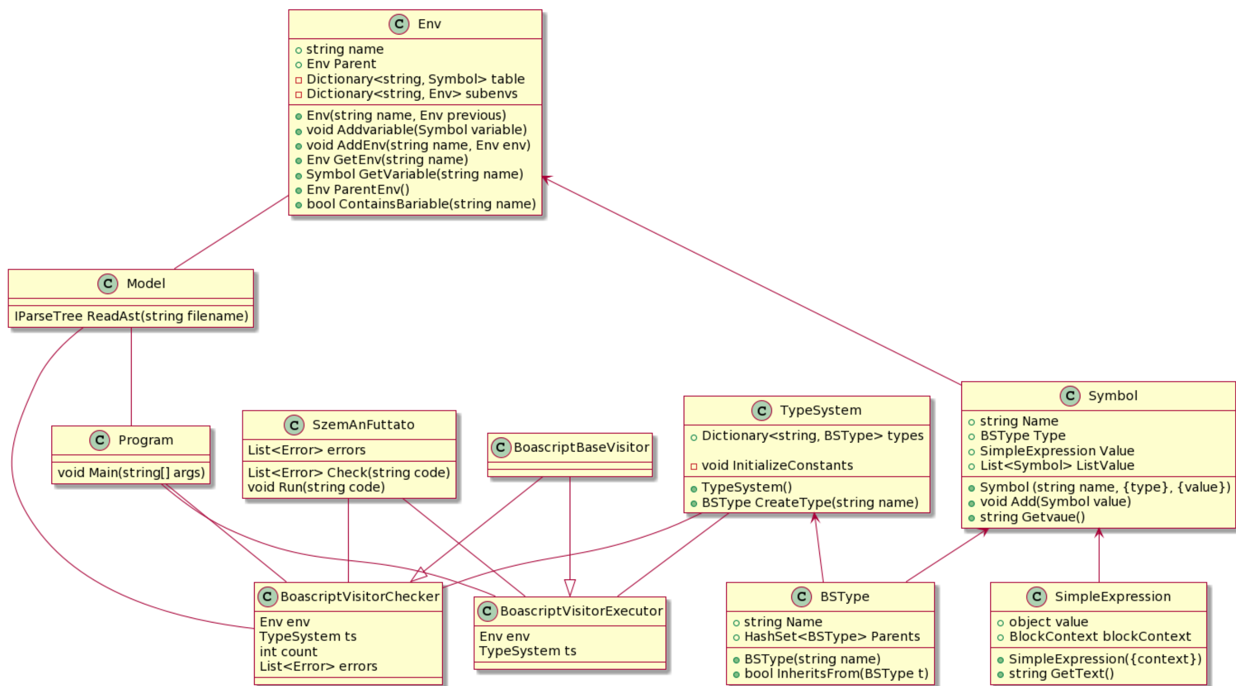
```
1 class auto{
2     var tavolsag : int = 2300
3     var benzin : float = 3.4
4 }
```

**18. ábra: Az osztályok szintaktikai felépítése**

# 4.2 Visitor

Az ANTLR által generált visitor felhasználásával (és kiterjesztésével) jártam be a szintaktikai elemzéskor létrehozott parse tree-t a „*visit*” metódusokkal a szemantikai elemzésnél, illetvekésőbb a forráskód futtatása is ezen keresztül történt.

Erre a feladatra létrehoztam két fájlt, mind a kettő a visitorból öröklődött és annak a függvényeit írja felül a feladatuk végrehajtására. A megoldás vázlata megtekinthető a 19. ábrán látható osztálydiagramon.




**19. ábra: A visitorokhoz tartozó osztály diagram részlet**

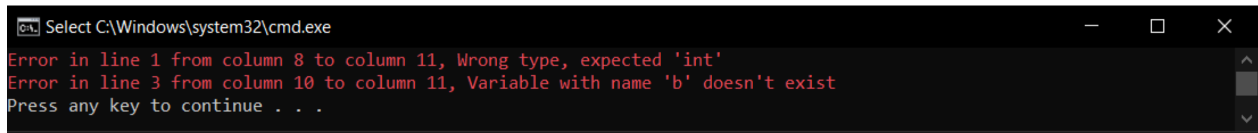
## 4.2.1 Ellenőrző

A Boascript működésének van egy része, amely egyedibbé teszi az interpretált nyelvek között. Ez pedig az, hogy bár a nyelv interpretált, mégis először egy ellenőrző visitor fut le, mely, ha talál egy hibát akkor, akárcsak egy fordított nyelv esetén, nem engedi a kódot futtatni. Ezt a célt szolgálja az *ellenőrző*, mely a kódhoz készült.

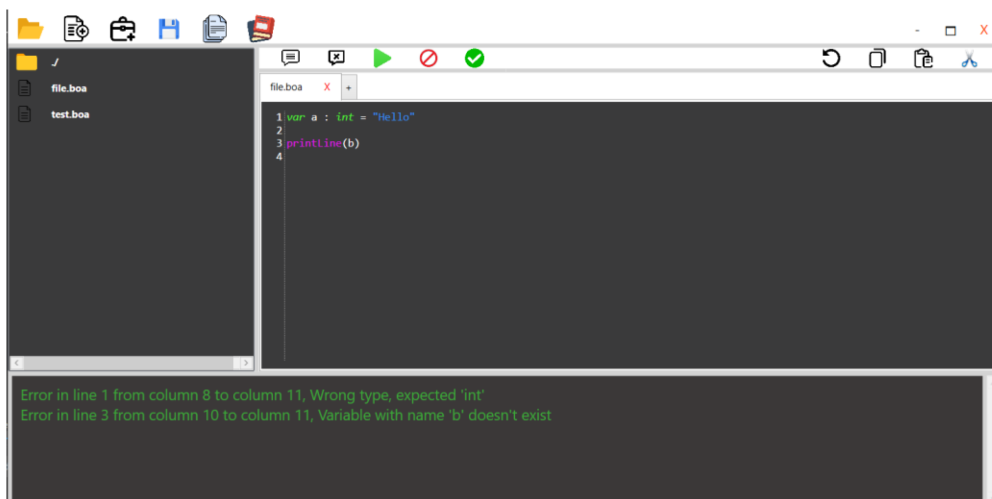
Alapvetően az ellenőrző építi fel a szimbólum táblát és ezzel együtt el is végzi a szemantikai elemzést. Abban az esetben, ha nem talált hibát, akkor lefuttathat a végrehajtó.

#### 4.2.1.1 Hibakezelés

Tekintve, hogy az ANTLR segítségével generált parser automatikusan elvégzi a szintaktikai hibakezelést, így azzal nem kellett komolyabban foglalkoznom, de a szemantikai elemzésben előforduló hibákat le kellet kezelnem valamilyen módon. Ezt a következő képpen oldottam meg: akárcsak egy szokványos fordított nyelvnél, így itt is végig fut az ellenőrző, még ha hibát talált, akkor is. Ezeknek a hibáknak a helyéről és a létéről visszajelzést tudunk adni a konzolon (*ábra 20*) és a felhasználói felületen is (*ábra 21*).




**20. ábra: Hibakezelés a konzolos környezetben**



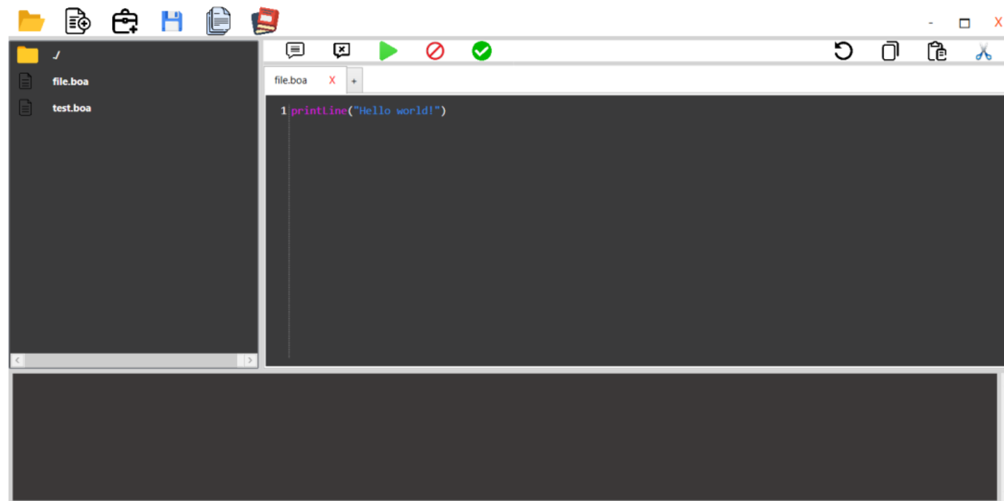

**21. ábra: Hibakezelés a grafikus környezetben**

### 4.2.2 Interpreter

A végrehajtó kód lényege, hogy a kódot értelmezze és sorról-sorra lefuttassa. Ezen a ponton már nem történik hiba ellenőrzés, a helyes működés esetén minden hibát meg kellett találnia az ellenőrzőnek. A végrehajtó kódban történik a vezérlési szerkezetek (mint például az elágazások) végrehajtása és a függvények kiértékelése is. Ennek köszönhetően lehetnek olyan elemek a korábban létrehozott szimbólumtáblában melyek értéke „*undefined*” a futtatás pillanatában, ugyanis ezeknek a változóknak az értéke (és akár még a típusa is) csak futtatáskor derül ki. Ezek elkerülése érdekében a beépített függvények visszatérési típusa előrre meg van határozva, így a futás idejű típus hibák lekorlátozódnak a felhasználó által létrehozott függvényekre.

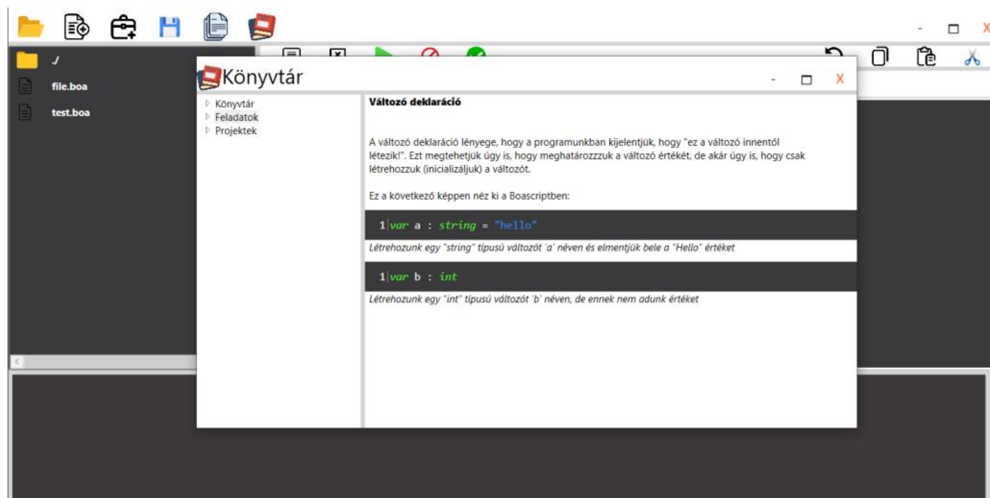
## 4.3 Grafikus keretrendszer

A C# WPF keretrendszerének felhasználásával hoztam létre a projekthez a mini-IDE-t. A mini-IDE-n keresztül valósulnak meg a dolgozatom igazi oktatás elméleti szinten levont következtetései. A grafikus rész elindításakor a felhasználót rögtön a szövegszerkesztő fogadja (*ábra 22*).



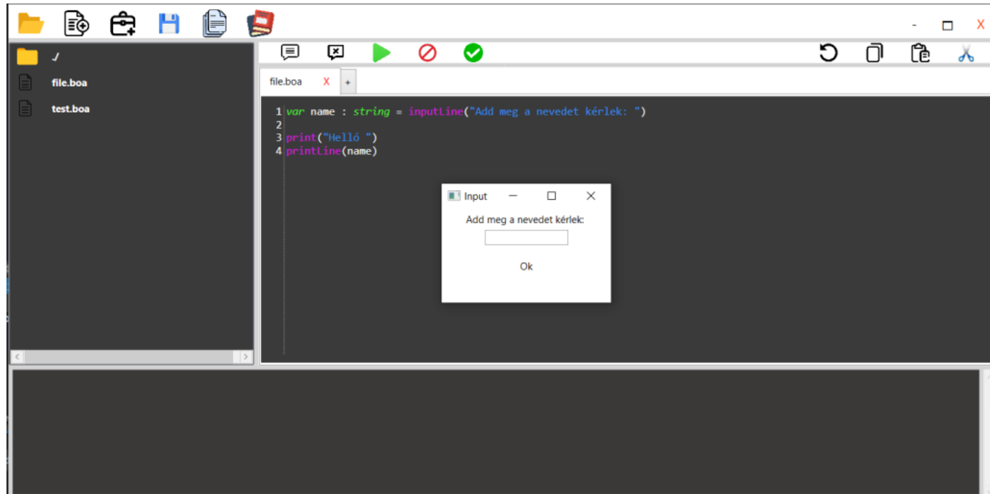

**22. ábra: A felhasználói felület**

A legfontosabb elem a szövegszerkesztő (ahova a Boascript kódot írjuk), de ezen kívül még több fontos funkció is megtalálható. Ilyen funkció például a beépített könyvtár, melyet a legfelső sorban található könyvekre kattintva érhet el a felhasználó (*ábra 23*) és amely tartalmazza a dokumentációt, a feladatokat és a projekteket.




**23. ábra: A beépített könyvtár felhasználói felülete**

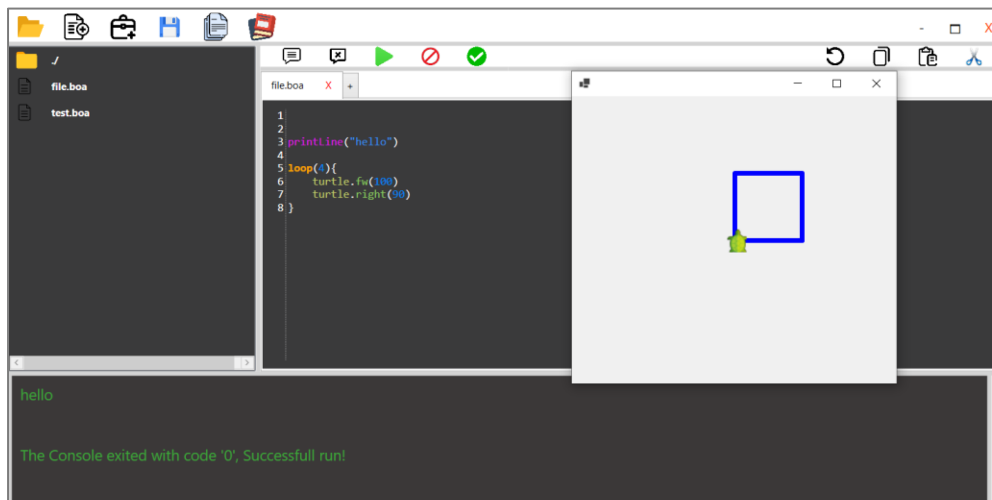
Érdekessége a Boascripthez tartozó mini-IDE-nek az is, hogy hogyan kezeli le az alapvetően konzolos utasításokat, mint például az „input”, amellyel alapvetően a konzolról lehet értéket beolvasni. Ezt egy InputPage (*ábra 24*) nevű ablak létrehozásával oldottam meg, mely az „input” vagy az „inputLine” utasítások esetén jelenik meg és lehetővé teszi az adatbekérést.




**24. ábra: Az InputPage**

Egy fontos eleme továbbá a Boascriptnek az, hogy a mini-IDE-be beépítve tartalmazza a teknőst, mint játékos rajzoló eszközt.

A teknős ablak (*ábra 25*) akkor csak jelenik meg, ha a felhasználó akármelyik „turtle.” metódust meghívja. A teknős parancsok nem zárják ki a konzolos elemek használatát, így a felhasználó több kimeneti módot is tud alkalmazni a kódjában.




**25. ábra: A teknős kód futtatásakor felugró ablak**

# 5. Szemléltető példa

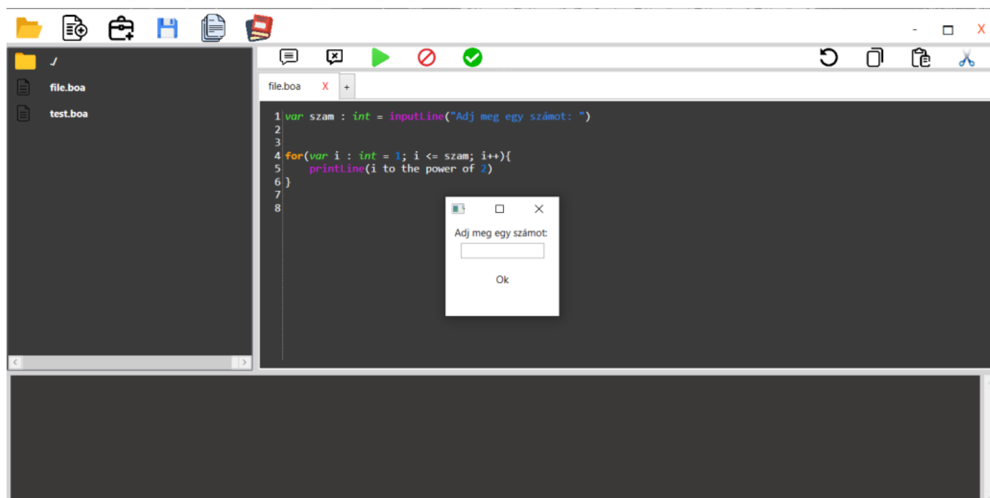
A következőkben egy egyszerű példán keresztül mutatom be az alkalmazás működését.

Tegyük fel, hogy a felhasználó akar írni egy olyan programot, ami bekér egy számot, majd ezt követően kiírja az első annyi négyzet számot mint amennyi a megadott szám.

```
1 var szam : int = inputLine("Adj meg egy számot: ")
2
3
4 for(var i : int = 1; i <= szam; i++){
5     printLine(i to the power of 2)
6 }
```

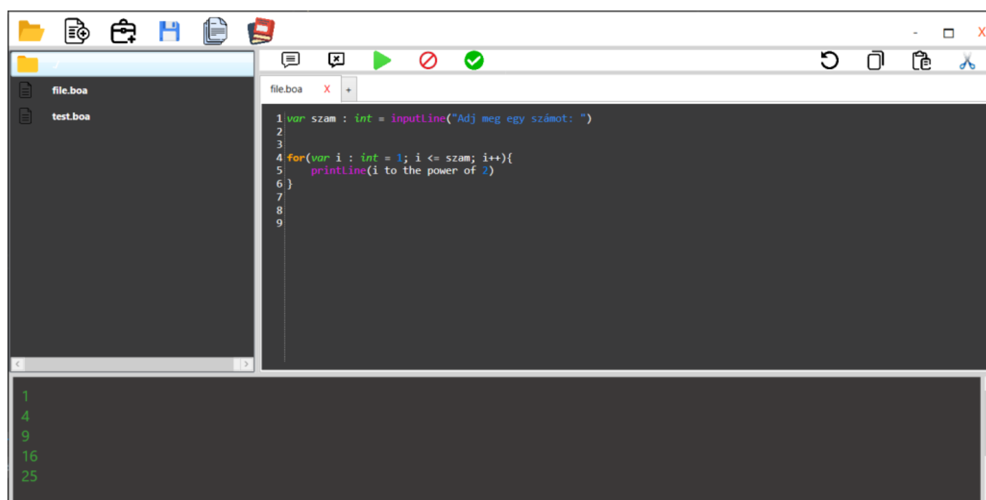
**26. ábra: A kód bemenete**

A felhasználó ezt követően a következőket tapasztalja: felugrik egy ablak, ami bemenetként kér egy értéket (*ábra 27*).




**27. ábra: A felugró ablak**

Majd miután azt megkapta, konzolra kiírja az eredményt (*ábra 28*).




**28. ábra: A program kimenete, ha a megadott szám „5”**

A program futtatásakor elkészül a szimbólumtábla (*ábra 29*), mely kettő értéket tartalmaz, a „szam” változót, melynek értéke a futásidőben derül ki és az „i” változót, melynek a kezdő értéke az 1.

| ID | Típus | Érték |
|---|---|---|
| Main scope | | |
| szam | int | {dinamikus} |
| i | int | 1 |

**29. ábra: A szemantikai elemzéskor elkészült szimbólumtábla**

# 6. Összefoglalás

Ezen dolgozat megírása közben jelent meg a hír, egyfajta módon megerősítve az itt leírtak fontosságát, hogy Ferenc pápa áldását adta egy olyan globális projektre, melynek célja a programozás oktatása [14]. Ezzel együtt az új tanrendben kötelezően beleépítették a, korábban közismereti informatikaként szereplő, digitális kultúra érettségibe a programozást már alapszinten. Napról-napra egyre nagyobb teret nyer a programozás a köztudatban, már olyanoknak is tudniuk kell programozni, legalább alapszinten, akiknek korábban a szakmájától az relatív távol állt (térképészek, biológusok, kémikusok), ehhez pedig fontos a megfelelő platform kiválasztása a tanuláshoz.

A dolgozatom megírása (és az ahhoz készített projekt megvalósítása) alatt nagyon sok tapasztalatot sikerült szereznem a programozás oktatás világában. Az irodalomkutatáskor meglepetten tapasztaltam a különböző módszereket, amiket mások találtak a programozás oktatás effektív felépítésére, de mégis egy valami van, amit semmilyen szoftver nem tud megváltoztatni, az pedig a programozás oktató tanár hozzáállása. Ebben is segített a dolgozat megírása, az itt megtanultakat (sikeresen) alkalmaztam programozás oktatóként is és az iskolámban diákként is jobb belátást kaptam arra, hogy hogyan kell egy korrepetálást helyesen levezetni, sőt, előfordult olyan is, hogy programozás tanárom kikérte a véleményemet amikor megismerte a dolgozatom témáját.

Ezzel együtt az általam készített projekt megvalósítása közben szerzett tapasztalat se elengedhető, amikor elkezdtem dolgozni rajta, még egészen tapasztalatlan voltam a Microsoft WPF használatában, azonban mégis sikerült egy átlátható felhasználó felületet összeépítenem. Azt sem szabad elfelejtenem, hogy a dolgozat előtt nem rendelkeztem túl sok tudással a fordítóprogramokat és értelmezőket illetően, ezt is sikerült orvosolnom, olyan mélyen bele ástam magamat a témába a dolgozat megírásakor, hogy rájöttem, ez egy olyan része a programozásnak, amivel később nagyon szívesen foglalkoznék többet is. Már most elkezdtem gyűjteni az ötleteket különböző, egzotikus, programozási nyelvre, amit meg szeretnék valósítani a szabadidőmben.

Összességében úgy látom, hogy sikerült egy olyan projektet készítenem, mellyel pozitívan gazdagítottam a saját tudásomon és a projektet szívesen fejleszteném tovább egy olyan projektté, mely akár egy nap a szakdolgozatom alapjait is alkothatja.

## 6.1 Továbbfejlesztési lehetőségek

Az elkészült nyelvet és mini-IDE-t többféleképpen lehetne továbbfejleszteni, ezekre néhány példa a következő:

- **Mesterséges intelligencia felhasználásával történő kód generálás/hiba felismerés**

  A mesterséges intelligencia nagyon sokat fejlődött az elmúlt években, egy egyszerűbb gépi tanuló algoritmus felhasználásával akár be is tudnám építeni a kódomba, mint egy hiba felismerő és akár még javító eszközt. Továbbá egy fejlettebb MI-t is fel tudnék használni arra, hogy a felhasználó által beírt szöveget automatikusan Boascript kóddá alakítsa.

- **A Boascript működésének átalakítása compileren alapuló működésre**

  A jelenlegi, interpreter alapú működés helyett lehetséges lenne kódot generálni. Ekkor a mini-IDE a generált kódot futtatná. Ezen megoldás lehetővé tenné a nyelv és mini-IDE élesebb elválasztását, de funkcionálisan nem jelentene javulást.

- **A nyelv beépített funkcióinak kibővítése**

  A nyelv egyelőre szinte teljesen az alapokra fekteti a hangsúlyt, ami bár kezdő programozóknak jó tud lenni, a későbbi fejlődést tudja lassítani. Ezért továbbfejleszteném a nyelvemet olyan „haladó” beépített elemekkel mint az öröklődés, a lambda függvények és a különböző osztályok.

- **(LEGO) robot programozás és egyéb gyerek- és oktatásbarát könyvtárak beépítése**

  A gyerekek és a kezdő programozók nehezen tudják meglátni, hogy mi értelme van annak a kódsornak, amit beírnak a számítógépbe. A programozást barátságosabbá és izgalmasabbá szeretném tenni olyan könyvtárak beépítésével (mint például a LEGO Mindstorms), melyek játékosabbá tudják tenni a tanulást.

## 6.2 Köszönetnyilvánítás

A dolgozat megírásában nagy segítségemre volt a konzulensem, Dr. Somogyi Ferenc Attila. Bár úgy tudom, hogy szakdolgozatban nem szokás köszönetét nyilvánítani a konzulensnek, de úgy érzem a TDK-s dolgozat ténye ezen változtat az én esetemben. Neki tartozok a legnagyobb hálával, az ő segítsége nélkül nem tudtam volna elkészíteni a dolgozatot, de a dolgozathoz készített projektet sem. Még szünidőben is, órák után rászánt időt, hogy a dolgozatomat megnézze, a kérdéseimre válaszoljon és konzultációs lehetőséget biztosítson számomra. Amikor az iskola levitte a hangulatomat, neki sikerült motiválnia arra, hogy folytassam tovább a munkát.

Szeretnék továbbá köszönetet mondani a családtagjaimnak is, akik a dolgozat megírása közben végig motiváltak és toltak előrre, akkor is, amikor meg akartam állni, amikor úgy éreztem, hogy „ez a TDK nem az én szintem”, akkor ők biztosítottak róla, hogy akkor is meg kell próbálnom. Alapvetően a nővérem, Lili, volt az, akinek a segítségével kitaláltam a Boascript elnevezést egy meleg nyári napon, munkaidőben üzeneteket váltva.

Ezen kívül szeretnék meg köszönetet mondani Kecskeméti Karinának, a középiskolám egyik programozás tanárának, akit nagyjából minden héten látogattam, akivel lehetőségem volt megtárgyalni a programozás oktatás elméleti kérdéseimet és időnként még egyfajta „gumikacsaként” is szolgált, neki elmagyarázva a kódom működését leltem rá fontosabb hibákra és egyszerűsítési módokra.

Végső, de nem utolsó sorban szeretném megköszönni a segítséget mind annak az anonim embernek, akik kitöltötték a kérdőívemet, ezzel is előre segítve a kutatásom folyamatát, illetve a nyári tábor alatt oktatott diákoknak, akik a tudtuk nélkül, megfigyeléseken keresztül járultak hozzá a kutatáshoz.

# Irodalomjegyzék


[1] Ó.-K. Viktor, „Boascript,” 31 Október 2023. [Online]. Available: https://github.com/Vikhun15/BoaScript. [Hozzáférés dátuma: 31 Október 2023].

[2] D. Dunning és J. Kruger, „Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments.,” *Journal of Personality and Social Psychology,* pp. 1121-1134, 1999.

[3] H. Felienne, „Hedy: A Gradual Language for Programming Education,” 13 Szeptember 2023. [Online]. Available: https://www.hedycode.com/. [Hozzáférés dátuma: 24 Október 2023].

[4] N. Winter, G. Saines, S. Erickson és M. Lott, „CodeCombat,” CodeCombat Inc., Február 2013. [Online]. Available: https://codecombat.com/. [Hozzáférés dátuma: 24 Október 2023].

[5] H. Partovi és A. Partovi, „Code.org,” [Online]. Available: https://code.org/. [Hozzáférés dátuma: 24 Október 2023].

[6] Ó.-K. Viktor, „Programozás tanulás kérdőív (Válaszok),” 7 Július 2023. [Online]. Available: https://docs.google.com/spreadsheets/d/1baYWrWkzNX1k6d4vxiqpGs4vUOQBnmeUieYjkyrCAFs/edit?usp=sharing. [Hozzáférés dátuma: 31 Október 2023].

[7] A. V. Aho, M. S. Lam és R. Sethi, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.

[8] R. Eisenberg, Sams Teach Yourself WPF in 24 Hours, Pearson Education, 2009.

[9] T. Parr, The Definitive ANTLR4 Reference, Pragmatic Publisher, 2013.

[10] R. H. R. J. J. V. Erich Gamma, Design Patterns, 1994.

[11] D. Grunwald, „AvalonEdit,” ICSharpCode, 19 Március 2023. [Online]. Available: https://github.com/icsharpcode/AvalonEdit. [Hozzáférés dátuma: 24 Október 2023].

[12] S. Nakov, „TurtleGraphics.NET,” 21 Február 2021. [Online]. Available: https://github.com/nakov/TurtleGraphics.NET. [Hozzáférés dátuma: 24 Október 2023].

[13] N. Hguyen, J. Garofalo és G. Bull, „Thinking about computational thinking,” *Journal of Digital Learning in Teacher Education 36:1,* pp. 6-18, 2020.

[14] H. Péter, „Ferenc pápa szeretné, ha több gyerek tanulna programozni,” 28 Október 2023. [Online]. Available: https://www.pcwplus.hu/pcwlite/ferenc-papa-szeretne-ha-tobb-gyerek-tanulna-programozni-348638.html. [Hozzáférés dátuma: 29 Október 2023].