EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Programozás struktogramokkal

Témavezető: Veszprémi Anna
Beosztás: mesteroktató

Szerző: Martin Dániel
Szak: programtervező informatikus BSc

Budapest, 2020

Tartalomjegyzék

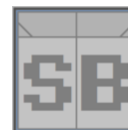
Tartalomjegyzék 1
Bevezető 3
- Mi az a struktogram? 3
- Főbb funkciók 4

Felhasználói dokumentáció 5
- A felhasználói felület 6
- Struktogramok építése 8
  - Szerkesztés 10
- Rekordok és konstansok 11
  - Rekordok 11
  - Konstansok 12
- Futtatás 13
  - Lépésenkénti futtatás 15
  - Felhasználói bemenet 16
- Indexelés 16
- Kinézet 17
- Beépített függvények 18
  - Kiírás és beolvasás 19
  - Tesztelés 21
- Exportálás 22
  - Képként 22
  - Az exportálás menete 23
- A programozási nyelv 24
  - Változók és értékadás 24
  - Vezérlési szerkezetek 26
  - Kifejezések 30
  - Direktívák 32
  - Alapértelmezett könyvtár 32
- Példák 33

Fejlesztői dokumentáció 34
- Grafikus felület 34
  - Témák 35
  - Canvas 35
- Virtuális gép 38

Programnyelv 39
Tervezési elvek 39
Formális nyelvtan 40
Az interpreter felépítése 42
Lexikális elemző 42
Parser 43
Konverter 44
Szintaxisfa és kifejezések 45
Az algoritmusok reprezentációja 45
StructoBlock 46
ConditionalStructBlock 47
InstructionBlock 48
MultiBranchBlock 48
SequenceBlock 49
SkipBlock 49
Kifejezések reprezentációja 50
Globális tárolók 52
GlobalFunctions 52
GlobalRecords 53
GlobalConstants 53
Exportálás 54
Bemenet 54
Előkészületek 55
Kimenet 56
Beállítások 57
Tesztelés 58
Nyelvi tesztek 58
Futtatási tesztek 58
Egyéb tesztek, megjegyzések 60
Debug-mód 61
Összefoglaló 62
Személyes megjegyzések 62

Bevezető



Egyetemi tanulmányaink során gyakran találkozunk struktogramokkal. Az előadásoknak és gyakorlatoknak egyaránt fontos eleme ez a programleírási modell. Magától értetődik, hogy szükség van egy szoftverre, mellyel interaktív módon lehet struktogramokat készíteni és futtatni. Ez a szoftver a *BlockCode*. Ezen dokumentáció ehhez a felhasználói szoftverhez tartozik. Két fő részre bontható:

- felhasználói dokumentációra, mely a felhasználónak biztosít útmutatást a szoftver használatához, illetve
- fejlesztői dokumentációra, mely a szoftver működésének mélyebb megértését és a programkód könnyebb átlátását segíti.

A szoftver elsősorban szakdolgozatnak készült, azonban bárkinek, aki informatikát tanul, hasznára válhat. Többek között ez a cél vezérelt a szoftver megírása közben.

A szoftver Windows operációs rendszerre íródott, és futtatásához mindössze a *.NET* keretrendszerre van szükség (legalább a 4.7.2-es verzióra), mely ingyenesen letölthető, és Windows 10 operációs rendszereken alapértelmezetten megtalálható. A szoftver gépigénye nem jelentős, így bármely PC-n, melyen legalább Windows 7 fut, probléma nélkül használható. Memóriahasználata 11-12 MB betöltés után, mely még komolyabb használat mellett sem megy fel jelentősen. Számos Linux-disztribúció alatt is fut a *BlockCode*, melynek feltétele az ingyenesen letölthető *Mono* keretrendszer.

Megjegyzés: mind a felhasználói, mind a fejlesztői dokumentáció feltételezi, hogy az olvasó legalább alapszinten jártas a programozásban, így az alapfogalmakat (mint pl. algoritmus, elágazás, változó) nem részletezi.

Mi az a struktogram?

Valószínű, hogy aki ezt a dokumentációt olvassa, már tisztában van a struktogram fogalmával, azonban nem árt összefoglalni a lényegét.

A struktogram a vizuális programozás egy fajtája, melyet elsősorban oktatási célokra találták ki. Az algoritmusok leírására „blokkokat” használ, melyeket egymás után lehet helyezni, és egymásba lehet ágyazni. A struktogramok – legalábbis az ebben a szoftverben megtervezhető struktogramok – általános, a népszerű nyelvekben mind fellelhető eszközöket biztosítanak az algoritmusok megtervezéséhez. Ilyen eszköz például az elágazás, ciklus, változó, függvény, kifejezések stb. A szoftver ezeket mind támogatja.

Noha a struktogramnak nincs hivatalos szabványa, számos konvenció született róluk. Ebben a szoftverben a struktogramok megvalósítása igyekszik a lehető legáltalánosabb lenni úgy, hogy az ELTE-n használt konvenciókat betartja.

Főbb funkciók

A szoftver a struktogramok tervezésén és eltárolásán kívül további funkciókkal is rendelkezik. Úgy, mint: A struktogramokat gyűjteménybe lehet szedni.

- Végre lehet őket hajtani *(futtatni)*, és közben követni a futásukat. A lépések naplózására is van lehetőség.
- Lépésenként is végre lehet hajtani a struktogramokat.
- Konstansokat és rekordokat is lehet definiálni, melyek a gyűjtemény részei.
- Ha az alapértelmezett (angol) kulcsszavak és operátorok nem felelnek meg a felhasználónak, módjában áll azokat lecserélni magyar vagy akár *egyéni* kulcsszavakra és operátorokra is.
- Ha több gyűjteményen keresztül is szüksége van a felhasználónak egy függvényre, rekordra vagy konstansra, definiálhatja azt egy közös állományban.
- A felület kinézetét témákkal meg lehet változtatni.
- A tömbök és stringek indexelésének bázisát szintén meg lehet változtatni.
- A megtervezett struktogramokat lehetséges *.jpg*, *.png* és *.bmp* formátumban exportálni, illetve C# és Python nyelvű programkóddá alakítani.

A szoftver mindegyik funkciója részletezésre kerül a következőkben mind felhasználói, mind fejlesztői szempontból.

Felhasználói dokumentáció

A *BlockCode* fő célja, hogy egyszerűen, különösebb ismeretek nélkül is lehessen struktogramokkal dolgozni, és azokat grafikus építőelemek *(ezentúl: blokkok)* egymásba ágyazásával, kényelmesen fel lehessen építeni. Habár ránézésre egyszerű a szoftver, természetéből adódóan érdemes áttekinteni a felhasználói dokumentációt, hogy tisztában legyen a szoftver lehetőségeivel és képességeivel. Az alábbi fejezetek tömören összefoglalják mindazt, amit a *BlockCode* használatához tudni érdemes. *Ez többek között a felhasználói felület kezelését, struktogramok interaktív építését és azok futtatását jelenti, és még sok mást.*

A szoftver Windows 10 alatt kerül bemutatásra, azonban működése, kinézete más operációs rendszereken is hasonló.

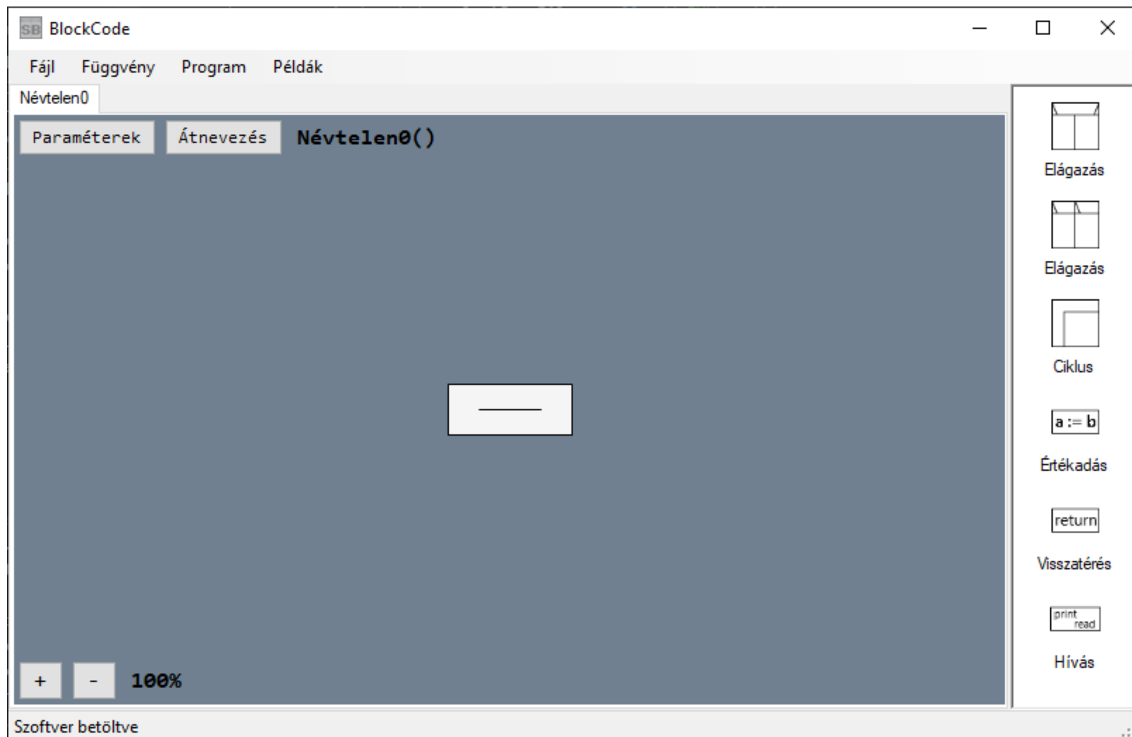
Windows-t használva a BlockCode.exe állománnyal lehet elindítani a szoftvert. Tartozik hozzá emellett még három állomány és egy mappa (*Language.dll*, *Settings.json*, *Standard.stpr* és *Examples*). Röviden összefoglalva funkcióik:

- **Language.dll** – a perzisztenciáért (betöltés, mentés) felelős könyvtár.
- **Examples** mappa – példakódok vannak benne.
- **Settings.json** – a szoftver beállításai.
- **Standard.stpr** – a szoftverhez mellékelt gyűjteményfájl, mely a *BlockCode* indításakor automatikusan betöltődik. Erről a dokumentáció *Programozási nyelv* című fejezetében olvashat. Neve: alapértelmezett könyvtár.
- **BlockScript.xml** – a szoftver működéséhez nem lényeges. A BlockScript.xml egy stílusfájl *Notepad++*-hoz.

A szoftver úgynevezett gyűjteményekkel dolgozik. A gyűjtemények kvázi projekteknek felelnek meg. Egybefoglalják mindazon függvényeket, rekordokat és konstansokat, melyek a felhasználó számára „összetartoznak”. Ezeket lehet menteni és betölteni. Indításkor egy új gyűjtemény jön létre, melyben egyetlen függvény található `Névtelen()` néven.

A felhasználói felület

Mivel a szoftver lényege a struktogramtervezés, így a felhasználói felület ezen funkció köré épül. Miután elindítja a szoftvert, az alábbi képernyő fogadja:

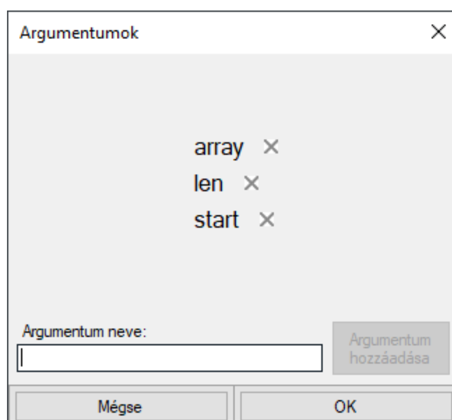


1. ábra – a főképernyő

A felület más *rendszerszintű* beállítások esetén eltérhet a fentitől.

A rögtön szembetűnő kék színű terület a vászon, melyen az éppen kiválasztott függvény látható és szerkeszthető. A vászon tetején két gomb és egy felirat látható. A **Paraméterek** gombbal a kiválasztott függvény formális paramétereit lehet megadni, ahogy az a 2. ábrán is látható. Itt például a szerkesztett függvény az *array*, *len* és *start* paramétereket kapja. Az **Argumentum hozzáadása** gombra kattintva adható hozzá a begépelt paraméternév a listához. (Lehet **Enter**t is használni a bevitelhez.) A paraméterek melletti kis × betűre kattintva az adott paraméter törölhető.

Az **Átnevezés** gombbal pedig a függvény nevét lehet megváltoztatni. A felirat a függvény szignatúráját tartalmazza (tehát nevét és formális paramétereit). A vászon alján a „+” és „–” gombokkal a struktogram betűméretét lehet beállítani. Mellettük látható a nagyítás mértéke. Egéren, görgetéssel is lehet nagyítani és kicsinyíteni, továbbá a numerikus billentyűzet „plusz” és „mínusz” gombjai is használhatók.



2. ábra – az argumentumszerkesztő dialógus

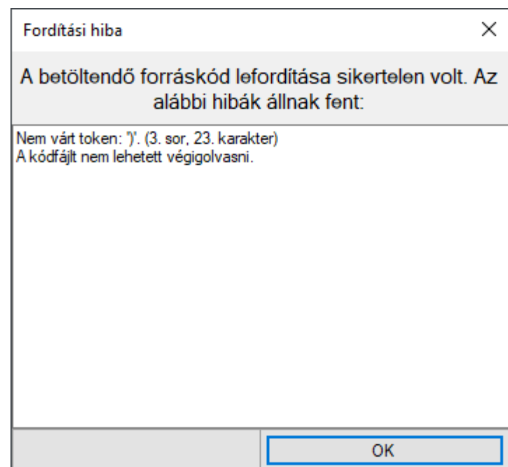
Megjegyzés: a struktogramok mérete csak a beállított betűmérettől függ, így a betűméret átállításával a méret is változik.

A vászontól jobbra hat ikon látható, melyek az egyes blokkokat reprezentálják. A használatuk a következő fejezetben kerül részletezésre.

Alul az állapotsáv látható. Itt állapotüzenetek jelennek meg, amik segítségével nyomon követheti, mi történik a szoftverben.

A vászon fölött található fülek mindegyike a gyűjtemény egy-egy függvényének felel meg. Ezekre kattintva lehet az adott függvényt kiválasztani. A képen például egy újonnan létrehozott gyűjtemény egyetlen függvénye található, melynek szignatúrája `Névtelen0()`. A függvény teste egy `skip;` utasítás. *Üres pedig elvből nem lehet egy függvény sem.*

Új függvényt a **Függvény** → **Új függvény** menüpont alatt lehet hozzáadni, melynek hatására a soron következő névtelen függvény jön létre. (Billentyűkombinációja a **Ctrl** + **F**.) Például ha az utolsó függvény neve „Névtelen4", akkor az új függvény neve „Névtelen5" lesz. Függvényt törölni ugyanebben a menüben lehet, a **Függvény** → **Függvény törlése** menüpontra kattintva (**Ctrl** + **Shift** + **F**). A törlés előtt a szoftver természetesen megerősítést kér.



3. ábra – a hibaüzenet-dialógus

Létező gyűjteményt (*.stpr* kiterjesztésű fájl) megnyitni a **Fájl** → **Megnyitás** menüpontra kattintva (**Ctrl** + **O**) lehet betölteni. A fájltallózó dialógusban válassza ki a betöltendő gyűjteményt! Ha a fájl hibás, tehát nem lehet betölteni, arról a 3. ábrán látható dialógusablakban hibaüzenetet kap.

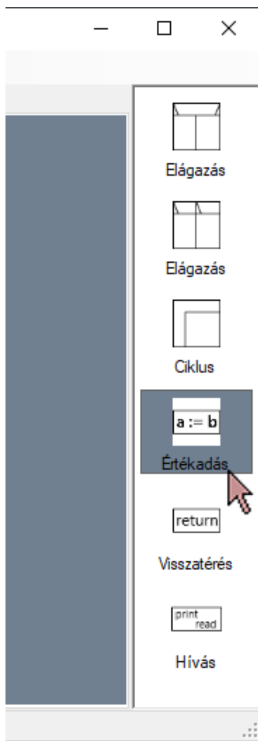
Gyűjteményt megnyitni nemcsak így lehet, hanem az *.stpr* fájloknak az ablakba való húzása és elengedése *(drag-and-drop)* által. Más kiterjesztésű fájlokra nem reagál a szoftver.

Gyűjteményt menteni a **Fájl** → **Mentés** menüponttal lehet, melynek hatására először egy fájltallózó dialógus jelenik meg. Itt adhatja meg a fájl mentési helyét és nevét. Ha már mentett gyűjteményt akar újramenteni (módosítani), akkor a fájltallózó dialógus nem jön elő. Ha más fájlba akarja menteni a gyűjteményt, kattintson a **Fájl** → **Mentés másként** menüpontra!

Új gyűjteményt a **Fájl** → **Új gyűjtemény** menüpontra kattintva hozhat létre vagy a **Ctrl** + **N** billentyűkombinációval. A létrehozás törli a jelenlegi gyűjteményt, így a szoftver megerősítést kér előtte. *Kivéve, ha alaphelyzetben van az aktuális gyűjtemény.*

A **Fájl** → **Beállítások** menüponttal (**Ctrl** + **Shift** + **S**) hívható elő a Beállítások ablak. Az itt szereplő összes opcióról a továbbiakban szó lesz.

A jobb oldali sávban hat különböző ikon található. Ezek mindegyike egy-egy blokktípust reprezentál. Az **első Elágazás** az IF-ELSE blokknak (kétágú elágazás) felel meg; a **második Elágazás** a többágú elágazásnak (IF-ELSEIF-ELSE). A **Ciklus**sal egy WHILE blokkot lehet létrehozni. Az **Értékadás**sal egy értékadásnak megfelelő utasításblokkot, **Visszatérés**sel egy RETURN-utasításblokkot, míg a **Hívás** ikonnal egy függvényhívást (szintén utasításblokk) lehet a struktogramhoz adni.

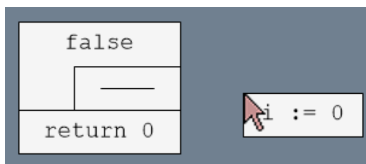


4. ábra

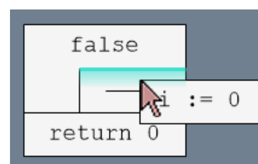
Egy új blokk hozzáadásához vigye az ikon fölé a kurzort (4. ábra), majd a bal egérgomb lenyomva tartása közben húzza a nagy vászonra (5. ábra)! Vonszolás közben láthatja az új blokkot a kurzor alatt.

Amint a struktogram egy blokkja fölé viszi a kurzort, egy megvastagított, árnyékot vető vonal mutatja a beszúrás helyét (6. ábra).

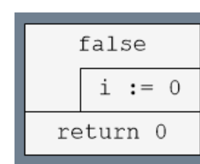
A blokk előtt/után megjelenő vonal azt jelenti, hogy a kijelölt blokk elé/után kívánja beszúrni az új blokkot. A két blokk közti vonal a két blokk közé való beszúrás szándékát jelöli. Ha a struktogram elé/után kívánja beszúrni a blokkot, vigye a struktogram alá/fölé a kurzort. A beszúrt elem a 7. ábrán látható.



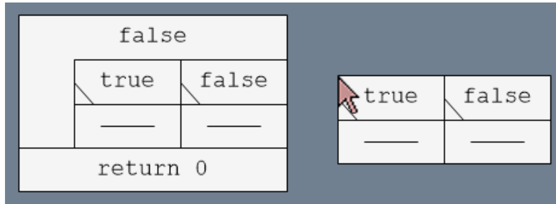
5. ábra



6. ábra

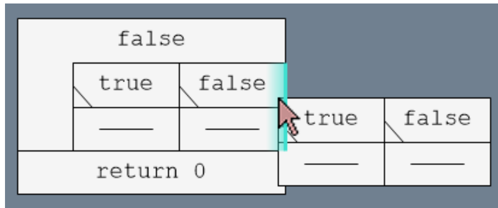


7. ábra



8. ábra

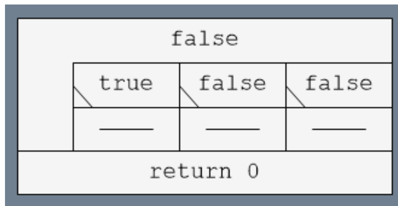
Többágú elágazás beszúrásakor egy speciális működés figyelhető meg. Ha az új blokkot egy már meglévő **többágú elágazás mellé** húzza (8. ábra), a beszúrást segítő vonal függőlegesen jelenik meg (9. ábra). Ekkor ha elengedi a blokkot, egy új ágat szúr be a már meglévő ágak mellé (10. ábra).



9. ábra

Egy már meglévő ág átmozgatása is hasonló módon történik; erről lejjebb.

Az egyes blokkokra bal egérgombbal duplán kattintva szerkeszthető azok „tartalma”. Erről részletesebben a *Programozási nyelv* fejezet *Vezérlési szerkezetek* alfejezetében olvashat.



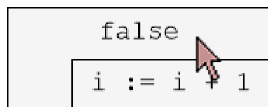
10. ábra

A struktogramokat nemcsak építeni lehet, hanem átrendezni is, vagy törölni belőlük egyes blokkokat. A blokkokat átrendezni – a bal egérgombot lenyomva tartva – vonszolva lehet. Miután egy blokkot kihúzott a struktogramból, két lehetősége van:

- Ha két blokk között, egy blokk előtt vagy után engedi el a blokkot, az oda beszúródik. A beszúrás helyét ugyanaz a „vonal" jelöli. Többágú elágazás ágait más ágak *mellé* is beszúrhatja.
- Ha a struktogramon kívül engedi fel az egérgombot, a vonszolt blokk törlődik.

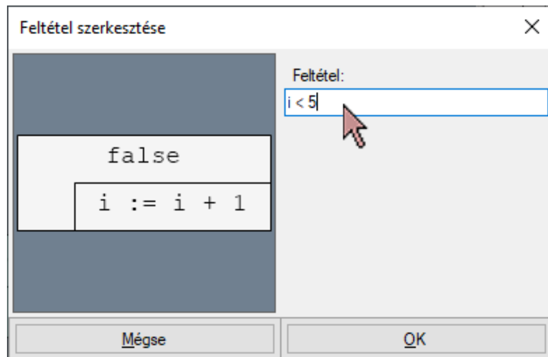
Figyelem: a blokkok törlése nem vonható vissza!

Szerkesztés

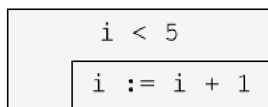


11. *ábra*

A strukrogramok építésének következő lépése a blokkok tartalmának (például egy ciklus feltételének) lecserélése. Egy adott blokk szerkesztését dupla kattintással (11. ábra) lehet megtenni, melynek hatására egy, az adott blokknak megfelelő szerkesztődialógus (12. ábra) ugrik fel. Ennek működéséről részletesen a *Vezérlési szerkezetek* alfejezetben olvashat. A balra látható példán egy WHILE ciklus feltétele cserélődik le az alapértelmezett `false`-ról `i < 5`-re (13. ábra).



12. *ábra*



13. *ábra*

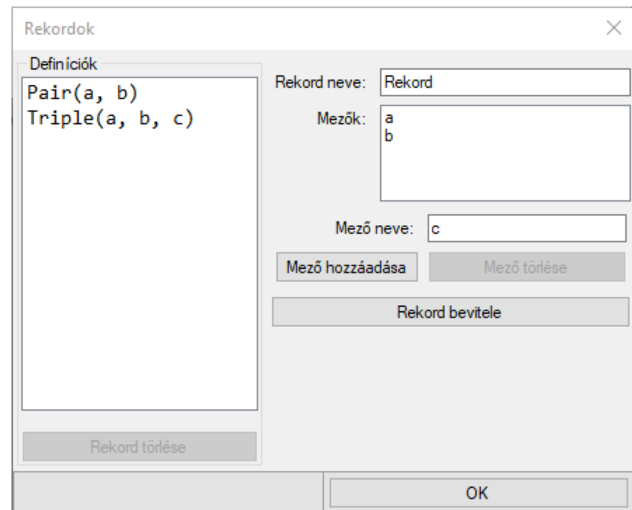
Rekordok és konstansok

Rekordok

Nem lenne teljes a szoftver rekordok nélkül. Gyakran előfordul, hogy összetett adatstruktúrát kell visszaadnia egy függvénynek. Ekkor elengedhetetlen a rekordok használata.

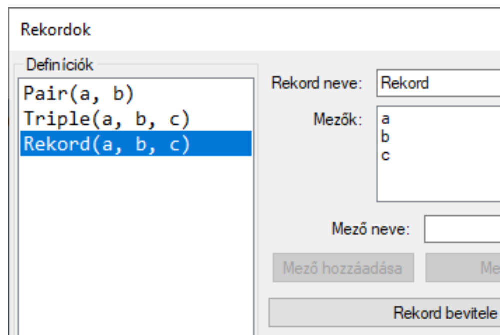
Mint a legtöbb nyelvben, úgy itt is ugyanaz a rekordok működési elve: a rekordnak van neve, és vannak mezői. Egy rekordhoz legalább egy mezőnek kell tartoznia. Új rekordot a grafikus felületen könnyedén lehet definiálni. A **Program → Rekordok** menüpontra kattintva (vagy a **Ctrl + R** billentyűkkel) elő lehet hívni a 14. ábrán látható rekordszerkesztő és -listázó dialógust.

Az illusztráción éppen egy új rekord kerül hozzáadásra „Rekord” néven, két mezővel – vagy ha rákattint a **Mező hozzáadása** gombra, három mezővel. A **Rekord bevitele** gomb segítségével adhatja hozzá a listához az új rekorddefiníciót. (Ennek eredménye lentebb látható. A **Mező törlése** gombbal a jobb oldali listában kijelölt mezőt törölheti. Ha egy már foglalt nevű rekordot próbál definiálni, a szoftver felajánlja, hogy lecseréli a már meglévő, azonos nevű definíciót az újjal. Csak a felhasználó által definiált rekordokat lehet módosítani.



14. ábra – a rekordszerkesztő dialógus

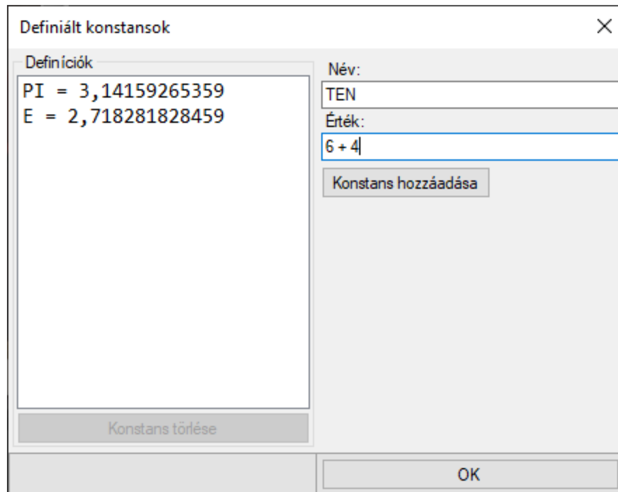
A bal oldali listában jelölheti ki szerkesztésre a kívánt rekordot. Az alapértelmezett könyvtárban definiált rekordokat nem lehet törölni, se szerkeszteni.



15. ábra – a „Rekord” nevű rekord szerkesztése

Egy rekordon belüli mezőre hivatkozni a kódban úgy kell, hogy a rekordot és a mező nevét egy ponttal kell elválasztani, akárcsak a legtöbb nyelvben. Például a *Bejegyzés* nevű rekord *ID* nevű mezőjére így lehet hivatkozni: `Bejegyzés.ID`. Új rekordpéldányt a `new` kulcsszóval lehet létrehozni, de erről később.

Egy másik, kevésbé létfontosságú, ám hasznos eleme a gyűjteményeknek a konstans. A konstans egy érték, melyhez egy „álnév” van rendelve, mellyel hivatkozni lehet rá. Természetesen felhasználói konstansokat is be lehet vinni a szoftverbe, amit a **Program → Konstansok menüponttal** előhívott dialógusablakban tehet meg, mely a 16. ábrán látható. (Billentyűkombinációja a **Ctrl + C**.)



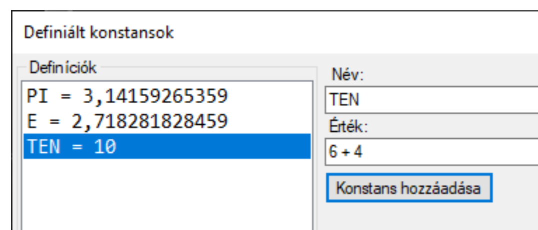
16. ábra – a konstansszerkesztő ablak

A konstans értéke nem feltétlenül valami elemi érték; akár összetett kifejezést is lehet a konstansnak adni, azonban maga a kifejezés nem marad meg, csupán annak értéke. Egy kifejezés konstans, ha annak értékét bármely struktogram lefuttatása nélkül meg lehet határozni. Tehát például az `abs(-1)` lehet konstans érték, de a `read() + 1` már nem lehet az.

Felhasználói függvényt is lehet használni a konstans kifejezésben. További részletekért olvassa el a *Kifejezések reprezentációja* fejezetet!

A kifejezés a **Konstans hozzáadása** gomb lenyomása után értékelődik ki és regisztrálódik, majd a bal oldali listában is megjelenik, ahogy az a 17. ábrán is látható.

Ügyeljen arra, hogy az érték megadása során ne használjon olyan kifejezést, melynek kiértékelése sok időbe kerül, vagy esetleg kiértékelése során „elszáll”, ugyanis az lefagyasztja a szoftvert.

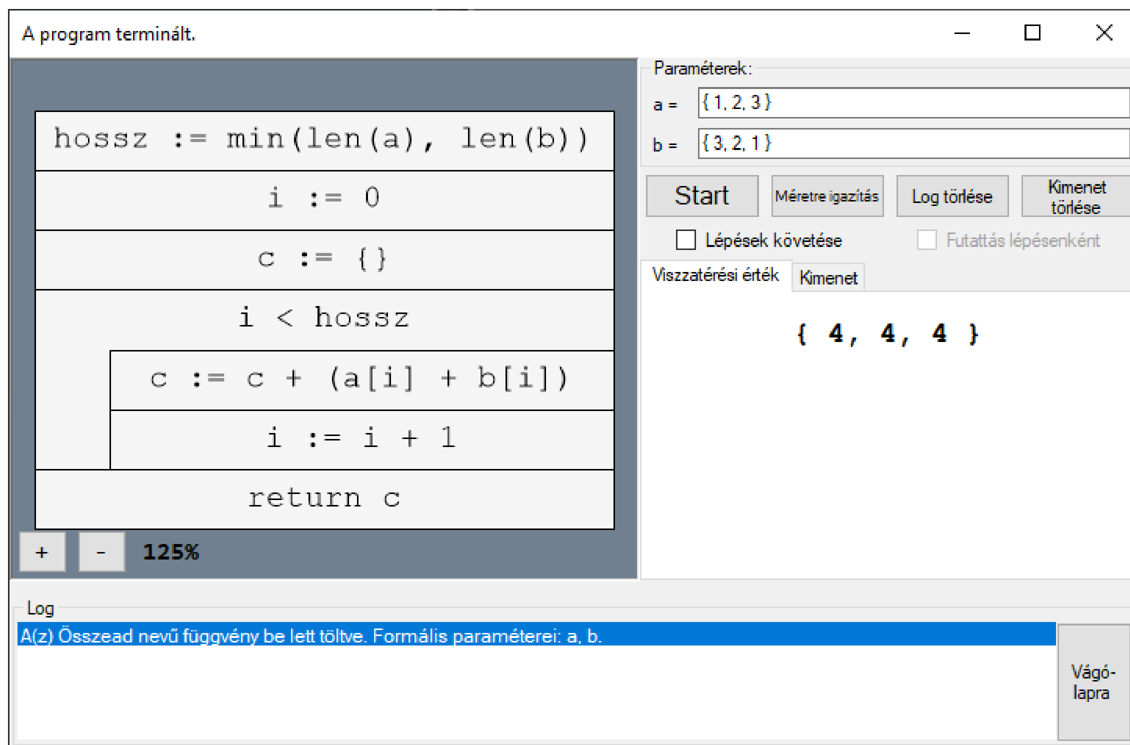


17. ábra – új konstans definiálva „TEN” néven

Futtatás

A *BlockCode* másik lényeges része egy virtuális gép, amelyben a megtervezett struktogramokat a megépítésük után azonnal futtathatja is, ahogy azt egy szkripttel tenné.

A futtatást a **Program → Futtatás** menüponttal előhívott **Program futtatása** nevű dialógusban teheti meg, melyet az **F5** billentyűvel szintúgy előhívhat. A dialógusablak – melybe példaképpen egy tömböket elemenként összegző függvény van betöltve – a 18. ábrán látható.



18. ábra – a programfuttató ablak

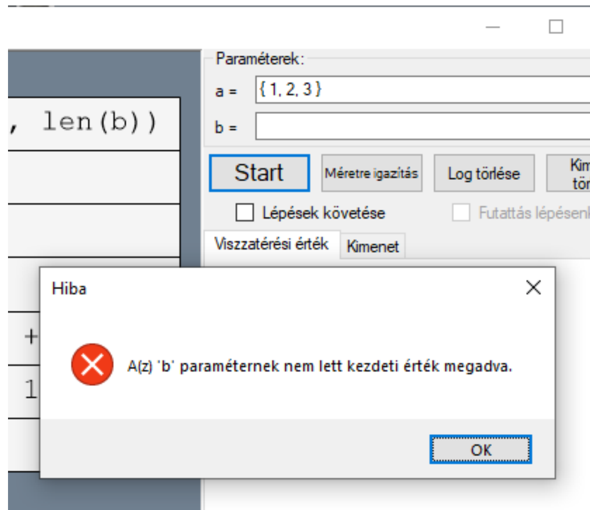
Az ablak három fő részre bontható:

A jobb felső doboz a **kezelő**. Ez felelős virtuális gépnek (és annak ki- és bemenetének) kezeléséért. Legfelül jelennek meg a függvény paramétereihez rendelt szövegdobozok, melyekkel azok értékét adhatja meg futás előtt. Középen a vezérlőgombok találhatók, lent pedig két fül látható: a **Visszatérési érték** fülön jelenik meg a függvény visszatérési értéke, vagy éppen a *„Nincs visszatérési érték."* felirat. A második, **Kimenet** nevű fülön az alapértelmezett (szöveges) kimenet található, melyre a `print()` utasítással lehet írni.

A bal felső dobozban jelenik meg az éppen futtatandó struktogram a szokásos **vásznon**. A struktogram méretét a kezelőben a **Méretre igazítás** gombbal igazíthatja a vászonhoz.

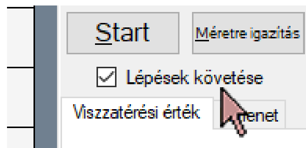
Az alsó doboz az úgynevezett **Log**. Itt jelennek meg állapotüzenetek a struktogram futtatása során, majd a futtatás befejeztével. A jobb oldali gombbal a log tartalmát a vágólapra másolhatja. A logot a kezelőben, a **Log törlése** gombra kattintva törölheti. A törlés a vágólap tartalmára persze nem hat.

A struktogram végrehajtását természetesen a **Start** gombbal lehet elindítani. Elindítás után ennek szövege **Stop**ra változik, amivel így pedig a futást lehet megszakítani. Az indítás előtt persze meg kell adni a függvény paramétereit a jobb felső sarokban. Ha nem adja meg az összes paramétert, a szoftver hibaüzenettel (19. ábra) tájékoztatja – akárcsak abban az esetben, ha hibás bemenetet ad meg. Paraméternek értelemszerűen konstans értéket kell megadni, akárcsak konstans definíciójánál. *Itt is ügyeljen arra, hogy a paraméterek kiszámolása belátható időn belül megtörténjen!*



19. ábra – hibás bemenet

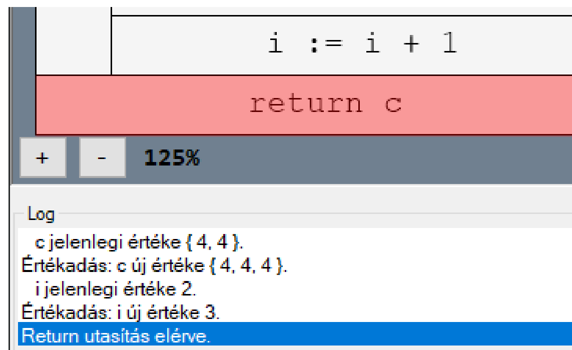
Ha a **Lépések követése** jelölőnégyzet (20. ábra) be van pipálva, akkor az éppen lefuttatásra kerülő blokk beszíneződik, ezzel jelezve a felhasználónak, hogy éppen hol tart a program futása. Továbbá a lépéskövetés közben a Logba is kerülnek bejegyzések, akárhányszor egy blokk végrehajtása megtörtént. Kivétel ez alól az értékadás, mely **előtt** is készül bejegyzés az értékadás bal oldalának jelenlegi értékéről, vagy éppen arról, hogy egy változó deklarálásra kerül. A bal oldalt látható a logolásra egy példa.



20. ábra

Az elkészült logbejegyzésekre kattintva (21. ábra) kijelölődik a struktogramon az a blokk, amely a bejegyzést okozta – persze csak akkor, ha a bejegyzést blokk okozta. Ez hibakereséshez hasznos lehet.

Természetesen ha a lépésenkénti futtatás aktiválva van, akkor is készülnek bejegyzések. Ennek leírása most következik.

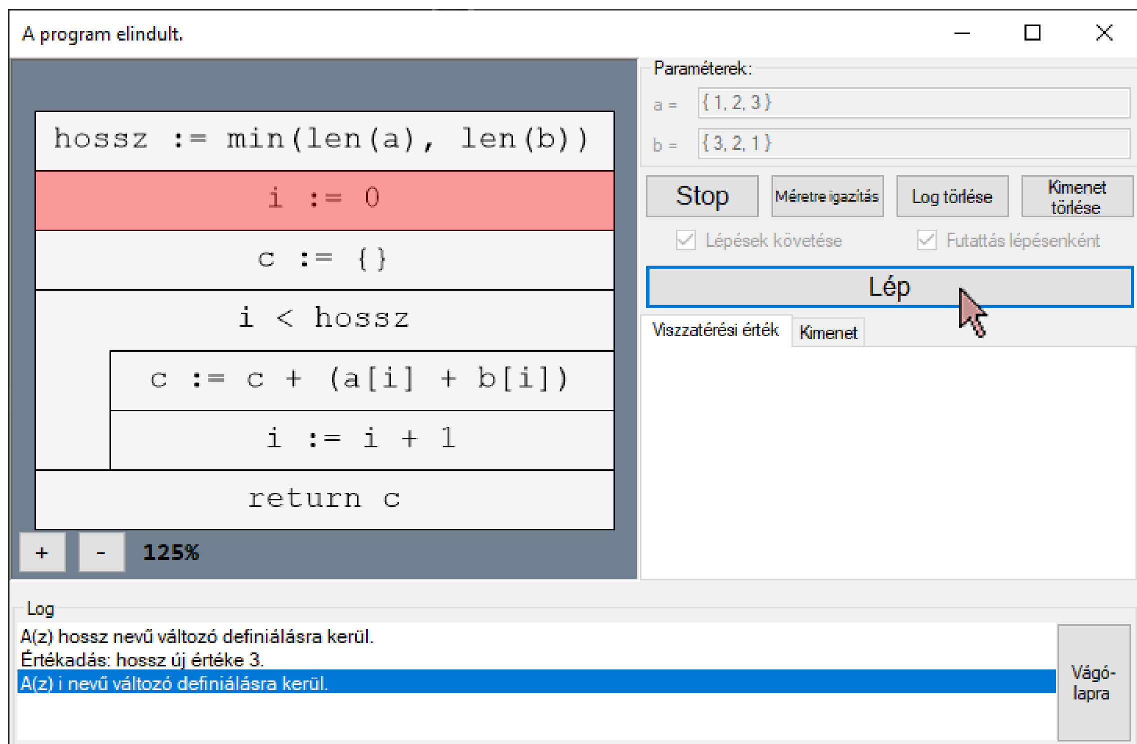


21. ábra

Lépésenkénti futtatás

A struktogramok futtatása oly' módon is lehetséges, hogy az egyes utasítások csak felhasználói kérésre futnak le. Ez a lépésenkénti futtatás. A **Futtatás lépésenként** jelölőnégyzet bepipálásával aktiválható. Ehhez persze a lépések követését is aktiválni kell. A két lépés lefuttatása közötti időt a **Beállítások → Utasításkésleltetés** pontban állíthatja be ezredmásodpercek formájában. 0 ms esetén nem történik késleltetés – még akkor sem, ha az előbb említett jelölőnégyzet be van pipálva. A maximálisan megadható időtartam 1 másodperc (1000 ms).

Bepipálás után előtűnik a **Lép** feliratú gomb. Ha lépésenkénti módban futtatja a struktogramot, ezzel a gombbal utasíthatja a virtuális gépet, hogy tegye meg az algoritmus következő lépését.

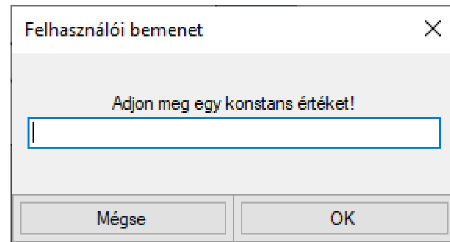


22. ábra – lépésenkénti futtatás

A 22. ábrán az első utasítás után áll a futás, és a **Lép** gomb megnyomása után a soron következő utasítás (jelen esetben az `i := 0` értékadás) fut le.

Felhasználói bemenet

Felhasználói bemenetet a `read()` függvénnyel lehet kérni. Amint a végrehajtás elér egy `read()` függvényt, megjelenik egy dialógus (23. ábra), melyben a promptüzenet és egy szövegdoboz látható. Az ide begépelt kifejezés az **OK** gomb vagy az **Enter** lenyomására kiértékelődik, és a `read()` függvény visszatérési értéke lesz. A függvények paramétert is lehet adni, mely az *„Adjon meg egy konstans kifejezést!”* felirat helyett jelenik meg.



23. ábra – a „bekérődialógus”

A `read()` függvény működéséről a *Kiírás és beolvasás* alfejezetben olvashat.

Indexelés

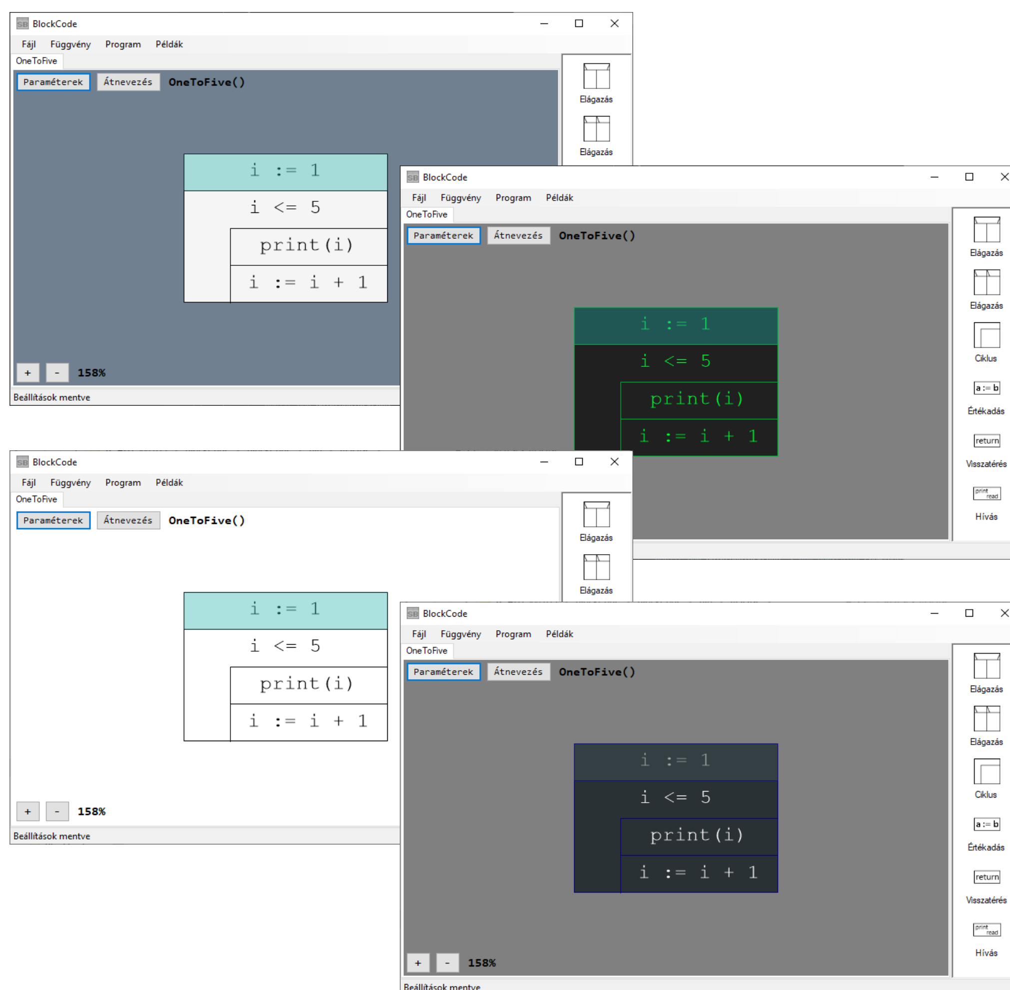
A beállításokban lehetőség van a tömbök és stringek indexelésének bázisát beállítani. Alapértelmezett beállításokon **nullától** történik az indexelés, de a **Fájl** → **Beállítások** → **Nullától való indexelés** opcióval be lehet állítani az egytől való indexelést is. Ez az opció a teljes gyűjteményre vonatkozik, és mentés során a fájlban direktívaként eltárolódik. A két lehetséges opció `#base 0` és `#base 1`. Az exportálás is ennek az opciónak figyelembevételével történik.

Kinézet

Lehetőség van a vászon és a struktogram színösszeállításának megváltoztatására *témák* segítségével. A beállított témát a **Fájl → Beállítások → Téma** opcióval lehet megváltoztatni. Jelenleg négy opció áll rendelkezésre:

- *alapértelmezett* – kék háttér, enyhén szürke struktogramszín, fekete betűszínnel, világoskék kijelöléssel.
- *sötét* – szürke háttér; sötétszürke, majdnem fekete struktogramszín, illetve klasszikus „konzolzöld” betűszín, világoszöld kijelöléssel.
- *fekete-fehér* – fehér a vászon és a struktogram háttere; a betűszín fekete, világoskék a kijelölés.
- *kék* – szürke a vászon, és a struktogram a kék árnyalatait használja. A szöveg fehér. A kijelölés szürkéskék.

Lassabb gépeken célszerű lehet a struktogramok renderelésének minőségét lejjebb venni. A renderelés minőségét a **Fájl → Beállítások → Jó minőségű** renderelés jelölőnégyzettel lehet szabályozni. Ez az opció a képként való exportálás minőségét nem befolyásolja; az mindig jó minőségű.



24. ábra – a négy téma (a fenti sorrendben)

Beépített függvények

Egy programnyelv sem teljes függvények nélkül. A *BlockCode* számos függvényt és metódust támogat, melyekkel matematikai műveletek, tömb- és stringmanipuláció, kiírás, beolvasás és tesztelés hajtható végre. Az alábbi listában az elérhető függvények kerülnek részletezésre:

- `abs(num)` – a megadott szám abszolútértékét adja meg.
- `floor(num)` – lefelé kerekíti a megadott számot.
- `round(num)` – a szokványos módon kerekíti a megadott számot.
- `rnd()` – véletlenszerű számot generál 0 és 1 között. (Az egy nincs a tartományban, de a nulla igen).
- `rnd(min, max)` – egy véletlenszámot (egész számot) ad vissza a megadott tartományból. A tartományba a határok is beleértendők.
- `min(a, b)` – visszaadja a két szám (vagy két string) közül a kisebbet. Stringeknél a már említett lexikografikus rendezés szerint dönt.
- `max(a, b)` – visszaadja a két szám (vagy két string) közül a nagyobbat. Stringeknél szintén az előbb említett lexikografikus rendezés szerint dönt.
- `len(seq)` – visszaadja a megadott tömb (vagy string) hosszát.
- `empty(seq)` – igaz pontosan akkor, ha `len(seq) = 0`.
- `sgn(num)` – visszaadja a megadott szám előjelét (-1, 0 vagy 1).
- `sqrt(num)` – megadja a szám négyzetgyökét.
- `sub(seq, start)` – visszaadja a tömb azon szeletét (vagy a string azon részstringjét), amely a `start` indextől kezdődik, és a tömb/string végéig tart.
- `sub(seq, start, len)` – visszaadja a tömb azon szeletét (vagy a string azon részstringjét), amely a `start` indextől kezdődik, és `len` hosszúságú.

Nem megfelelő típusú paraméterek, érvénytelen paraméterek, túl sok paraméter vagy túl kevés paraméter esetén futási idejű hiba keletkezik, melyről a felhasználó dialógus formájában kap tájékoztatást.

A `print()` metódus az alapértelmezett kimenetre ír egy sort, mely a metódusnak paraméterként megadott kifejezések értékét írja ki formázva vagy formázatlanul. Kétféle paraméterezéssel hívható meg:

- Ha az első argumentum string, akkor a `print()` behelyettesítő módon működik. A string úgy kerül kiírásra, hogy a stringben található helyjelölők lecserélődnek a második, harmadik, … argumentummal. A helyjelölők formátuma `{0}`, `{1}`, …; ahol a számok a második, harmadik, … argumentumokat számozzák meg. Nem szabad elfelejteni, hogy egy string csak akkor kezelődik formázó stringként, ha dollárjellel ($) kezdődik.
  Példa a használatra:

  ```
  print($”{0} kedvenc száma a {1}.”, ”Béla”, 42);
  ```

  Ekkor a kimenet:

  ```
  Béla kedvenc száma a 42.
  ```

  A formázó stringben szereplő helyjelölők bármilyen sorrendben szerepelhetnek, tehát ha a fenti stringben a `{0}` és `{1}` felcserélhető. Ha kevesebb argumentum van megadva, mint ahány helyjelölő, az hibát eredményez. Ha több, akkor a „többletek” helyére üres string kerül.

- Ha az első argumentum nem string, akkor az argumentumok vesszővel és szóközzel elválasztva íródnak ki.
  Például:

  ```
  print(42, { 1, 2, 3 }, true, ”alma”);
  ```

  Ekkor a kimenet:

  ```
  42, { 1, 2, 3 }, igaz, alma
  ```

Minden kiírt sor végére kerül egy sortörés, ami univerzálisan az ’\r’ és ’\n’ karakter. Amennyiben argumentumok nélkül hívják meg a metódust, csupán egy üres sor íródik ki.

A `read()` függvény felhasználói bemenetet kér egy felugró ablakon keresztül, ahogy az a *Felhasználói bemenet* alfejezetben látható. A `read()` függvény visszatérési értéke a dialógusban megadott kifejezés értéke. Ha a felhasználó a **Mégse** gombra kattint vagy bezárja a dialógust, a program futása megszakad.

A függvény kétféle módon hívható meg:

- Ha egy argumentuma van (ami egy string kell hogy legyen), akkor azt üzenetként jelenik meg a dialógusablakban.
  Például:

  ```
  x := read(”Adjon meg egy számot!”);
  ```

  Miután a felhasználó megadott egy számot, az az `x` változónak értékül adódik. Persze ez még önmagában nem garantálja, hogy `x` csak szám típusú lehet, hiszen a típus csak futási időben derül ki.

- Ha nincs argumentuma, az alapértelmezett üzenet jelenik meg a dialógusban.

  ```
  x := read();
  ```

A dialógusban természetesen csak konstans kifejezést lehet megadni. Ez azt jelenti, hogy a kifejezést futtatás nélkül is ki lehet értékelni, azaz nem tartalmazhat sem változót, sem nem-konstans függvényt.

Példa a *Felhasználói bemenet* alfejezetben, a 23. ábrán látható.

A `read()` függvény dinamikusan típusos jellege miatt nem exportálható.

A megtervezett struktogram dinamikus tesztelésére használható a `test()` metódus, amely az alábbi két módon használható:

- Két argumentum esetén a metódus kiértékeli mindkét argumentumot, és egyenlőségvizsgálatot végez. A vizsgálat eredményét az alapértelmezett kimenetre írja. A lehetséges értékek a „Sikeres teszt.” és a „Sikertelen teszt. (…)”, ahol az zárójelben a kapott érték és az elvárt érték szerepel. **Tömbök esetén elemenkénti egyenlőségvizsgálatot végez** a megszokott módon.
  Például:

  ```
  test(2 + 3, 5);
  test(2 + 4, 5);
  ```

  Ennek eredménye a kimeneten:

  ```
  Sikeres teszt.
  Sikertelen teszt. (5 helyett 6.)
  ```

- Egy argumentum esetén azt vizsgálja a metódus, hogy történik-e az argumentum kiértékelése során hiba (kivétel). A teszt tehát akkor sikeres, ha történik hiba, illetve ha nem történik hiba, a teszt sikertelen.
  Például:

  ```
  test(”A” * ”B”);
  test(”A” + ”B”);
  ```

  Ennek eredménye a kimeneten:

  ```
  Sikeres teszt.
  Sikertelen teszt. (Nem történt kivétel.)
  ```

A metódus jelenleg nem exportálható, azaz a struktogramban használt `test()` hívások nem kerülnek az exportálás által előállított kódba; Python nyelven a helyükre *pass* utasítás kerül.

Exportálás

A szoftver lehetővé teszi az elkészített struktogramok (pontosabban az egész gyűjtemény) konvertálását „valódi” programozási nyelvekre. Jelenleg két nyelvre lehet exportálni: C# és Python nyelvre. Az exportálás során a betöltött gyűjtemény teljes tartalma exportálásra kerül – beleértve a függvényeket, rekordokat és konstansokat. Viszont az alapértelmezett könyvtárban definiált függvények, rekordok és konstansok közül csak azok kerülnek ki, melyek az adott gyűjteményben fel vannak használva. A beépített függvények nem külön függvények formájában kerülnek exportálásra, hanem a kimeneti nyelv „alapértelmezett könyváraiban” elérhető függvényekre fordulnak le.

Az exportált gyűjtemény egyetlen fájlba kerül, mely C# esetén *.cs* kiterjesztésű, míg Python esetén *.py* kiterjesztésű). A fájl nevét a gyűjtemény neve adja, melyet a **Program → Gyűjtemény átnevezése** menüpont alatt lehet megváltoztatni. A név eltárolására a mentési fájlban a fájl elején lévő `#name` mező szolgál. Ha nincs név megadva, a „Névtelen” nevet kapja a fájl. Természetesen az exportálás során kézzel is meg lehet adni a fájl nevét.

Az exportált fájlok UTF-8 kódolásúak (BOM nélkül), akárcsak a mentési fájlok. A mentés a kód bonyolultságától függően sokáig is eltarthat. Közben a szoftver felülete deaktiválva van. Az exportálás befejeztét az állapotsoron megjelenő üzenet mutatja.

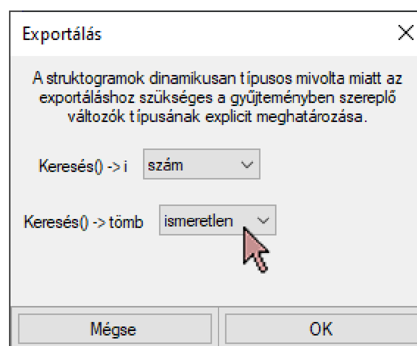
A struktogramok dinamikusan típusos mivolta miatt a gyűjtemények exportálása nem mindig egyértelmű. A változók és paraméterek típusát nem mindig lehet a meglévő információk alapján eldönteni, és így nem lehet konkrét kódot előállítani. Ennek a problémának áthidalásához felhasználói bemenetre van szükség, melyet a szoftver egy dialógusablakban kér be az exportálás előtt. Erről van szó a következő alfejezetben.

Képként

A programozási nyelvek mellett lehetőség van az éppen aktuális struktogram képként való exportálására. Ehhez kattintson a **Fájl → Exportálás → Képként** menüpontra, majd a felugró mentési ablakban adja meg a kép nevét és fájlformátumát. A mentési ablakot a **Ctrl + E** gyorsbillentyűvel is előhívhatja. A mentett kép mérete a nagyítástól, színösszeállítása a beállított témától függ. Formátuma lehet *.jpg*, *.bmp* és *.png*.

Mentés előtt érdemes lehet a nagyítani vagy éppen kicsinyíteni a struktogramot, ugyanis a nagyítástól függően változhat annak szerkezete.

A megnyitott gyűjtemény exportálásához a **Fájl → Exportálás** menüben válassza ki az exportálás kívánt nyelvét! Az exportálás előtt megjelenik az a bizonyos dialógusablak (25. ábra), melyben meg kell adnia a gyűjteményben (és esetlegesen a standard könyvtárban) használt változók és formális paraméterek típusát úgy, hogy a mellettük lévő listából kiválasztja azt. Ez jobb oldalt van illusztrálva.



25. ábra – az exportablak

Mint az látható, a `Keresés()` függvény `i` változójának típusát ki tudta következtetni magától a program; a `tömb` paraméter típusát azonban nem, így azt manuálisan kell kiválasztania a legördülő listából.

A listában az alábbi opciók jelennek meg:

- *ismeretlen* – a változó vagy paraméter típusát nem lehet meghatározni. Ha bármelyik legördülő listában ez van kiválasztva, a gyűjteményt nem feltétlenül lehet helyesen exportálni.
- *szám* – ez lebegőpontos számot jelent alapvetően, de bizonyos esetekben az exporter képes rájönni, ha használhat egész típust is. C# esetén *double*-ként (vagy egész szám esetén *int*-ként) fog exportálódni.
- *egész szám* – tömbök és stringek indexeléséhez ezt érdemes választani. C#-ban *int* típusra fordul le. Python esetén nincs ez és az előző között különbség.
- *logikai* – értelemszerűen logikai változókhoz és paraméterekhez kell társítani. C# esetén *boolean* típusra fordul le.
- *string* – szöveges típus. C# esetén *string* típusra fordul.
- *tömb* – olyan tömböt jelent, melynek elemtípusa nincs meghatározva. C# esetén *List<dynamic>* típusra fordul le.
- Továbbá a gyűjteményben és a standard könyvtárban definiált rekordok nevei is megjelennek itt.

A függvények visszatérési értékét viszont nem szükséges megadni, ugyanis ha benne minden változónak és paraméternek a típusa meg van határozva, azt a szoftver magától is képes kikövetkeztetni. A rekordok mezőinek típusa nincs meghatározva. C# esetén azok mind *dynamic* típusúak.

Fontos figyelembe venni, hogy az exportálás nem garantálja, hogy az exportált program azonnal futtatható lesz. Még akkor sem, ha minden változó típusa meg van határozva. Így előfordulhat, hogy az elkészült kódot még helyenként át kell írni ahhoz, hogy fusson. A szintaktikai helyesség viszont garantált.

A programozási nyelv

Habár a szoftver célja, hogy a felhasználók tisztán grafikusan, struktogramokkal programozhassanak, a megtervezett programokat valamilyen formában el is kell tárolni. Erre a szoftver a kifejezetten erre a célra készített *BlockScript* programozási nyelvet használja. Ennek a nyelvnek a részletezése következik.

A részletezendő nyelv erősen, dinamikusan típusos programozási nyelv. A legtöbb programozási nyelvhez hasonlóan támogatja az imperatív és procedurális programozási elveket, azaz a program definiálása egy utasítássorozat megadásával történik, melyeket függvényekben lehet megadni. Az objektumorientált programozást, funkcionális stílusú programozást és pointereket jelenleg nem támogatja. Az összetett adatstruktúrák közül a tömböt és a rekordot támogatja. Konstanst lehet benne definiálni.

Kommentet is lehet benne hagyni, melyek (//) dupla perrel kezdődnek, és a sor végéig tartanak. Ezek a gyűjtemény újramentésekor elvesznek.

Megjegyzés: a BlockScript nyelv működésének ismerete nélkül is használható a szoftver, azonban érdemes megismerkedni a nyelvvel.

Változók és értékadás

A változók explicit deklarálására – akárcsak a struktogramok esetében – nincs szükség. A deklaráció a változó első értékadásakor történik meg. Ha a deklaráció nem történik meg, a változó használata esetén futási idejű hiba keletkezik.

A nyelv erősen típusos, azaz egy változó első értékadásakor dől el, hogy mi lesz a változó típusa. Ez később nem változtatható meg. Ha egy deklarált változónak más típusú értéket akarunk adni, mint amilyen típusú, az is hibát eredményez.

Implicit típuskasztolás vagy -konverzió nem történhet meg, kivéve akkor, amikor egy nem string típusú értéket stringként akarunk használni. Ez később kerül részletezésre.

A változók értékadása hasonlóan történik, mint a legtöbb nyelven. Minden értékadás egy bal és egy jobb oldalból áll. A bal oldalon álló kifejezés értékét változtatja meg az értékadás. Háromféle bal oldala lehet egy értékadásnak annak típusa szempontjából:

Változó

A változók névvel ellátott memóriaterületek, és mint a legtöbb nyelvben, értéküket itt is meg lehet változtatni. A *BlockScript* négyféle változótípust támogat:

- **szám** – egész vagy tört szám. Értéke $\pm 5\times 10^{-324}$ és $\pm 1{,}7\times 10^{308}$ közé eshet.
- **tömb** – elemek sorozata. Elemei meg vannak számozva (nullától „hossz - 1”-ig, vagy egytől „hossz”-ig). Ez az *indexelés*, melynek segítségével egy tömb bármely elemét meg lehet változtatni. Ha nem létező elemre hivatkozik (alul- vagy túlindexel), az futási idejű hibát okoz. A *BlockScript*ben – a legtöbb nyelvvel ellentétben – a tömbök elemtípusa vegyes is lehet akár. Függvényhíváskor referencia szerint adódnak át.

- **string** – karakterliterál. Függvényhívás esetén érték szerint adódnak át. Egy string lehet formázó vagy nem-formázó. A formázó stringek dollárjellel ($) kezdődnek. Ezek lényegéről a *Kiírás és beolvasás* alfejezetben esett szó. Az itt használt stringek nem használnak kilépési karaktereket, tehát speciális karakterek nem lehetnek a stringben. Stringeket noha (a tömbökhöz hasonlóan) lehet indexelni, azok egyes karaktereit megváltoztatni **nem** lehet. *(A stringek immutábilisak BlockScript-ben.)*
- **logikai érték** – értéke *igaz* vagy *hamis* (`true` vagy `false`). Elemi logikai műveleteket lehet rajtuk végezni.
- **rekord** – összetett adattípus. Függvényhívás során érték szerint adódik át. Az egyes mezői első értékadáskor jönnek létre; addig nem elérhetők. Típusuk is ekkor dől el. A rekordok a `new` operátorral példányosíthatók.

Tömbelem és rekordmező

Sem a tömbök, sem a rekordok nem immutábilisak, így nemcsak egyben lehet őket megváltoztatni, hanem azok mezőit/elemeit is külön-külön, értékadáskor rájuk hivatkozva.

Az értékadások szintaxisa a következő alfejezetben olvasható.

Vezérlési szerkezetek

A *BlockCode* és a *BlockScript* nyelv az alapvető vezérlési szerkezeteket – a szekvenciát, elágazást és ciklust – mind támogatja. Ezek *BlockScript*-beli szintaxisa, illetve *BlockCode*-beli használata következik ebben az alfejezetben.

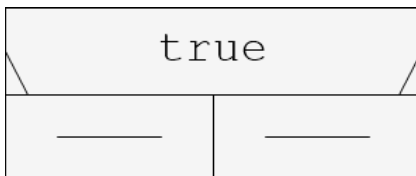
Elágazás (IF-THEN-ELSE)

A nyelvben van lehetőség kétágú *if* elágazás készítésére az alábbi szintaxissal:

```
if FELTÉTEL then
   „IGAZ ÁG”
end if
```

```
if FELTÉTEL then
   „IGAZ ÁG”
else
   „HAMIS ÁG”
end if
```

Feltételnek csak olyan kifejezést lehet megadni, melyhez lehet igazságértéket rendelni. A nyelv erősen típusos mivolta miatt a csak logikai értékekre kiértékelhető kifejezések minősülnek állításnak, tehát számértékekhez nem rendelhető igazságérték (azaz például `1 = true` állítás érvénytelen).



26. ábra – kétágú elágazás

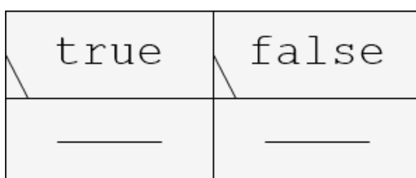
Az újonnan létrehozott elágazás feltétele `true`, és mindkét ága egy-egy `skip;` utasítás. A 26. ábrán látható. A feltételre (felső sáv) duplán kattintva (bal gombbal) megjelenik a feltételszerkesztő ablak, ahová egy kifejezést írva, majd az **OK** gombot vagy az **Enter** lenyomva lecserélheti a feltételt.

Elágazás (IF-THEN-ELSEIF-ELSE)

A fenti szerkezet mindig pontosan két ágat tartalmaz. Ha kettő ágnál többre van szükség, az alábbi szintaxist lehet rá használni. Ez a többágú elágazás.

```
if FELTÉTEL then
   „IGAZ ÁG”
else if FELTÉTEL then
   „KÜLÖNBEN-ÁG”
end if
```

```
if FELTÉTEL then
   „IGAZ ÁG”
else if FELTÉTEL then
   „KÜLÖNBEN-ÁG”
else
   „HAMIS ÁG”
end if
```



27. ábra – többágú elágazás

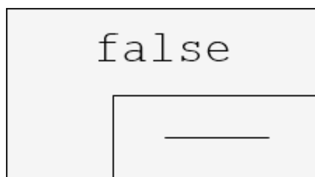
A 27. ábrán oldalt látható a többágú elágazás kezdeti állapotában. Két ága van, melyek feltétele `true` és `false`. Mindkét elágazás teste egy-egy `skip;` utasítás. Az ágak tetején lévő feltételre kattintva ugyanaz a dialógus jelenik meg a feltétel szerkesztésére, mint a „sima” elágazás esetén.

Ciklus (WHILE)

A nyelv csak a WHILE ciklust ismeri, tehát nincs lehetőség a FOR (és egyéb) ciklusok használatára. Ennek szintaxisa az alábbi:

```
while FELTÉTEL do
   CIKLUSMAG
next
```

Az újonnan létrehozott ciklus a 28. ábrán látható. Feltétele `false`, és teste egy `skip;` utasítás. A feltételre duplán kattintva lehet azt szerkeszteni, akárcsak az elágazásokat.



28. ábra – ciklus

A feltételre persze itt is ugyanaz igaz, ami az *if* elágazás feltételére. A ciklusból explicit utasítással (break) nem lehet kilépni, ugyanis azt a programozók rendszerint nem tekintik „helyes” vezérlési elvnek.

Függvény- vagy metódusdefiníció

```
function NÉV(ARGUMENTUMLISTA)
    FÜGGVÉNYTEST
end function
```

Az argumentumok (formális paraméterek) vesszővel vannak elválasztva. Ha egy „függvénynek” nincs visszatérési értéke, az metódusnak minősül. Metódusok nem szerepelhetnek kifejezésben; ez esetben a program hibát jelez. Függvényeket önmagukban meghívni nem lehet.

Rekorddefiníció

```
record NÉV(MEZŐLISTA)
```

A mezőnevek vesszővel vannak elválasztva. Nullamezőjű rekordot nem lehet definiálni; az hibát okoz.

Konstansdefiníció

```
const KONSTANSNÉV := KIFEJEZÉS;
```

Bár nyilvánvaló, azért érdemes megemlíteni, hogy a jobb oldalon szereplő kifejezésnek konstans kifejezésnek kell lennie, tehát változót nem tartalmazhat, és függvényt sem hívhat.

A konstansdefiníció végén lennie kell pontosvesszőnek, a rekorddefiníció végén viszont nem.

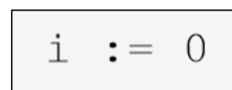
Utasítások

- értékadás

  Az első értékadás egyben változódeklarációnak is minősül.

```
KIFEJEZÉS := KIFEJEZÉS;
KIFEJEZÉS := { ELEMLISTA };
KIFEJEZÉS.MEZŐNÉV := KIFEJEZÉS;
KIFEJEZÉS[KIFEJEZÉS] := KIFEJEZÉS;
```

  Kezdeti állapotában az értékadás-blokk az `i := 0` kifejezést tartalmazza. Duplán rákattintva megjelenik a szerkesztőablak, ahol az értékadás bal- és jobb oldalát is szerkesztheti. A bal oldal csakis *balkifejezés* lehet. A 29. ábrán látható az értékadás.



29. ábra – értékadás

- metódus- vagy függvényhívás

```
FÜGGVÉNYNÉV(PARAMÉTERLISTA);
```

  A *BlockScript* nem tesz különbséget metódus és függvény között, akárcsak a legtöbb nyelv, így a függvényeket is ugyanúgy lehet hívni, akárcsak a metódusokat. A visszatérési érték ekkor figyelmen kívül hagyásra kerül. Az argumentumokat vesszővel kell elválasztani.

30. ábra – függvényhívás

  A függvényhívás kezdetben egy `print()` utasítást tartalmaz, mely egy üres sort ír az alapértelmezett kimenetre. Dupla kattintással szerkeszthető. A felugró dialógusablakban csak függvényhívást adhat meg kifejezésképpen. A 30. ábrán látható.

- **visszatérés**
  Visszatérni függvényből lehet az alábbi szintaxissal:

```
return;   return KIFEJEZÉS;
```

Ha a függvény csak metódus *(nincs visszatérési értéke)*, akkor az abból való visszatérés esetén a `KIFEJEZÉS` mező üres. A program *pontosan akkor* tekint egy „függvényt” függvénynek, ha szerepel benne olyan `return`, ahol a `KIFEJEZÉS` nem üres.

31. ábra – visszatérés

A visszatérési blokk kezdetben (31. ábra) egy `return 0` utasítást tartalmaz, mely szintén dupla kattintással lecserélhető.

- **skip**
  Nem hajt végre változtatást. Kvázi helytartóként szolgál, hogy az elágazások, ciklusok és függvények ne legyenek üresek.

```
skip;
```

Grafikus reprezentációja (32. ábra) egy vízszintes vonal. Manuálisan nem szúrható be; csupán helytartóként funkcionál. Nem szerkeszthető, nem törölhető; automatikusan cserélődik le.

32. ábra – skip

Megjegyzés: az utasítások végére pontosvesszőt kell tenni.

Kifejezések

A programkódban az utasítások mellett szerepelnek kifejezések is, amelyek egy értéket határoznak meg, fejeznek ki. A kifejezések konstans értékekből, operátorokból, zárójelekből és függvényhívásokból állhatak. Fontos megjegyezni, hogy a kifejezéseknek nincs mellékhatása, azaz bármiféle tárolt érték megváltoztatása csak értékadás útján történhet.

A nyelvben számos operátor van definiálva. Ezek használata úgy történik, mint a legtöbb programozási nyelvben.

Az `=`, `!=` operátorok működése intuitív. Csak azonos típusú értékeken alkalmazhatók, azaz nem történik implicit típuskonverzió. Ha az operandusok típusa szám, logikai érték, string vagy rekord, akkor érték szerinti egyenlőség- vagy egyenlőtlenségvizsgálat történik. Tömbök esetén a vizsgálat referencia szerinti. Rekordok esetén a mezők kerülnek összehasonlításra. Stringek esetén karakterenkénti vizsgálat történik, a kis- és nagybetűk között különbséget téve. Az operátorok visszatérési értéke logikai.

A `>`, `>=`, `<`, `<=` operátorok csak számokon és stringeken alkalmazhatók. Számok esetén a megszokott rendezést használja. Stringek esetén ábécé szerinti lexikografikus rendezést használ a program, a rendszeren beállított alapértelmezett kódolás szerint, a kis- és nagybetűk között különbséget téve. Az operátorok visszatérési értéke logikai.

Az `in` operátor kétféle módon használható:

- *érték* `in` *tömb* – igaz pontosan akkor, ha a bal oldali érték szerepel a tömbben.
- *string* `in` *string* – igaz pontosan akkor, ha a bal oldali string „részstringje” a jobb oldalinak.

Az `and`, `or`, `xor`, `imp`, `not` operátorok az elemi logikában is megtalálható operátoroknak felelnek meg. A `not` operátor egyoperandusú. Mindegyik operátor operandusainak típusa logikai. Az `a imp b` kifejezés implikációs, és pontosan a `not a or b` kifejezésnek felel meg.

A `new` operátor speciális. A segítségével lehet új rekordot létrehozni. Egyetlen operandusa a rekord neve. Visszatérési értéke pedig az újonnan létrehozott *üres* rekord. A rekord mezői a *pont/szelektor* operátorral érhetők el. Jobb oldalt látható egy példa a használatra.

```
r := new Rekord;
r.mező := érték;
print(r.mező);
```

Az *előjel* operátor számokon használható csak. Értelemszerűen az adott érték (-1)-szeresét adja vissza.

A `^`, `/`, `div`, `mod` operátorok szintén csak számokra alkalmazhatók. A `^` operátor a hatványoperátor. A `/` operátor a „hagyományos”, azaz maradék nélküli osztás. A `div` operátor a maradékos osztás, míg a `mod` operátor a maradékképzés. Természetesen a visszatérési érték is szám.

A + operátor többféle módon is használható. Hagyományosan két szám összeadására használható, azonban az alábbi használati esetek is megengedettek:

- *string* + *érték*; *érték* + *string* – a jobb vagy bal oldali értéket stringgé alakítja, majd a másik operandussal konkatenálja azt. A visszatérési érték string. Bármely érték stringgé alakítható. A logikai értékek stringgé alakítása az éppen beállított kulcsszókészlettől függ.
- *tömb* + *tömb* – visszaadja a bal oldali és a jobb oldali tömb másolatának konkatenáltját.
- *tömb* + *érték*; *érték* + *tömb* – visszaadja a megadott tömb másolatát úgy, hogy a végére vagy elejére konkatenálja a megadott értéket.

A – operátor szintúgy többféle módon is használható a hagyományos kivonáson kívül:

- *string* - *szám* – leveszi a megadott string végéről a jobb oldali operandusban megadott számú karaktert. Maga a string itt sem módosul.
- *tömb* - *érték* – visszaadja a tömb másolatát úgy, hogy a megadott elem nem szerepel benne. Ha a megadott elem többször is szerepel a tömbben, akkor annak mindegyik előfordulása eltűnik.

A * operátor a hagyományos szorzás mellett további használati esetekkel is rendelkezik:

- *tömb* * *szám*; *szám* * *tömb* – a megadott tömb elemeit megadott alkalommal ismételi, majd az elemeket tömbként visszaadja. A szorzószámot lefelé kerekítve értelmezi. Negatív szám futási idejű hibát okoz. A nullával való szorzás üres tömböt eredményez.
- *string* * *szám*; *szám* * *string* – az előző használati esethez hasonlóan működik, viszont itt a be- és kimenet string. Negatív szám futási idejű hibát okoz. A nullával való szorzás üres stringet eredményez.

Ha az operátoroknak nem megfelelő típusúak az operandusai, az futási idejű hibát eredményez, melyről a felhasználó tájékoztatást kap.

Az operátorok precedenciája (*műveleti sorrendje*) a C nyelvben használt precedenciákhoz hasonló.

Operátor	**Leírás**	**Precedencia**
`.`	rekordmező kiválasztása	0
`(…), […]`	függvényhívás és indexer	1
`not, -, new`	negálás és negatív előjel	2
`^, *, /,` `mod, div`	hatvány, szorzás, osztás, maradékképzés és maradékos osztás	3
`+, -`	összeadás és kivonás	4
`>, >=, <, <=`	relációs jelek	5
`=, !=, in`	„egyenlő” és „nem egyenlő”	6
`and`	„és”	7
`or, xor`	„és” és „kizáró vagy”	8
`imp`	implikáció	9

A hatványoperátor és az implikáció jobbasszociatív, vagyis az `a ^ b ^ c` jelentése `a ^ (b ^ c)`, és az `a imp  b imp c` jelentése `a imp (b imp c)`.

Direktívák

A kódfájlok a programkódon kívül tartalmazhatnak direktívákat is, melyek a fájl elején találhatók. Ezek metainformációval szolgálnak a kódról, és az egész gyűjteményre vonatkoznak. Minden direktíva egy kettőskereszttel (#) kezdődik, utána a direktíva neve következik, majd az argumentuma, mely a sor végéig tart. Minden direktívának pontosan egyetlen argumentuma van.

A három lehetséges direktíva:

- *#base* – azt határozza meg, hogy a gyűjtemény honnan indexeli a tömböket és stringeket. Lehetséges argumentuma a 0 és 1. Más értékek hibához vezetnek.
- *#name* – a gyűjtemény neve. Sortörésen kívül bármilyen karakterekből állhat.

  #set – a gyűjtemény által használt kulcsszókészletet határozza meg. Az alapértelmezett készletet a *„default”* jelenti. A magyart a *„hungarian”*, míg az egyénit a *„custom”*. Más argumentumok hibához vezetnek.
  Megjegyzés: az egyéni kulcsszókészlet a Settings.json fájlban található.

Ha ugyanaz a direktíva többször is szerepel a kódfájlban, akkor csak az első kerül feldolgozásra; a többit figyelmen kívül hagyja a preparser. A fenti listában nem szereplő direktívákat figyelmen kívül hagyja.

Alapértelmezett könyvtár

Alapesetben a *BlockCode* mellett található a *Standard.stpr* fájl, melynek tartalma a szoftver indításakor betöltődik – feltéve, hogy a **Fájl → Beállítások → Alapértelmezett könyvtár betöltése** opció be van pipálva. A fájlban (*BlockScript* nyelven) lehetnek rekordok, konstansok és függvények is. Exportálás esetén ezek közül csak azok kerülnek a kimeneti fájlba, amelyeket a betöltött gyűjtemény használ.

Az alapértelmezett könyvtárban nincsenek direktívák, ugyanis nincs neve, az indexelés mindig nullától történik, illetve az alapértelmezett kulcsszókészletet használja.

Példák

A szoftver mellé számos gyűjtemény is mellékelve van, melyek példák útján mutatják be annak funkcióit. Ezeket a **Példák** menüből választhatja ki. Egy adott példára kattintva az betöltődik. Az alábbi lista röviden összefoglalja, hogy az egyes példák „mire jók”:

- **Bemenet és kiírás** – a `read()` és `print()` függvények használatát mutatja be.
- **Keresés** – egy függvény, amely a megadott tömbből visszaadja az első páratlan számot.
- **Kifizetés** –a forintban megadott összeget címletek lineáris kombinációjára bontja fel.
- **Kockadobás** – megadott számú kockadobást generál.
- **LNKO és LKK** – kiszámolja a megadott számok legnagyobb közös osztóját és legkisebb közös többszörösét.
- **Minimum- és maximum** – egy tömb elemei közül visszaadja az (első) legkisebbet és (első) legnagyobbat.
- **Prímek** – kiszámolja egy megadott számig az összes prímszámot.
- **Rekurzív függvények** – az összegzés, minimum- és maximumkeresés mutatja be rekurzív megközelítéssel.
- **Összegzés** – összeadja a megadott tömb elemeit.
- **Betűk cseréje** – lecserléi a magyar ábécé minden magánhangzóját E betűre a bemeneten.
- **Mátrixok** – mátrixokkal (azokat tömbökkel reprezentálva) kapcsolatos műveleteket mutat be.

További példákat is lehet készíteni, melyeket az *Examples* mappába – vagy annak egy almappájába lehet tenni. A szoftver újraindítása után megjelennek a **Példák** menüben az újonnan hozzáadott példák.

Fejlesztői dokumentáció

A felhasználói dokumentáció összefoglalta a szoftver képességeit, annak használati eseteit. A fejlesztői dokumentáció kitér arra, hogy az eddig leírtakat mi is teszi lehetővé. Összefoglalja mindazt, amire annak van szüksége, aki a *BlockCode* forráskódját olvassa. De a nevével ellentétben persze nem csak fejlesztőknek való a szakdolgozat ezen része.

A forráskód *Visual Studio Community 2017*-ben íródott. Minden, ami a programkódhoz kötődik, egy úgynevezett solution-ön belül található, melynek *BlockCode* a neve. A solution tartalmazza a szoftver projektjeit. A kód három projektre lett osztva:

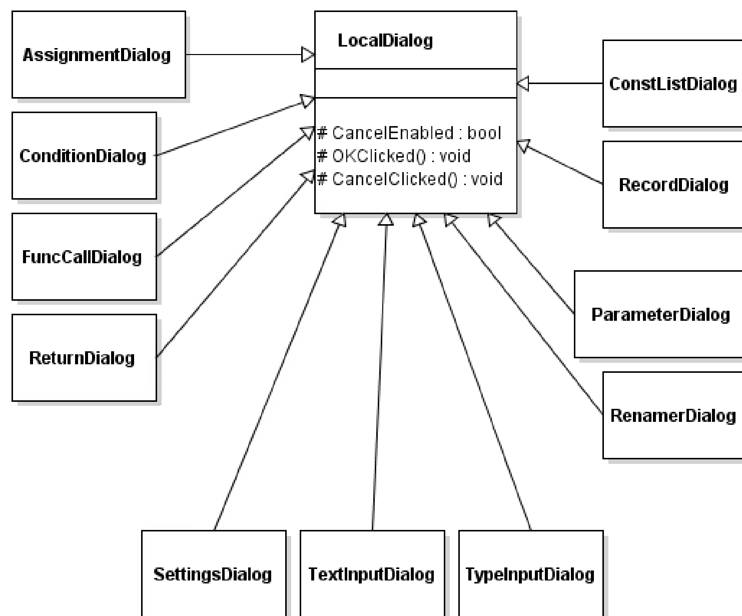
- **BlockCode** – ez maga a szoftver. Annak modelljei, nézetei, belső reprezentációi.
- **Language** – ez a *BlockScript* nyelv értelmezésére írt könyvtár.
- **Tester** – ez pedig a másik két projekt tesztelésére készült. Egységteszteket tartalmaz.

A következő fejezetekből kiderül mindhárom projekt működése. Minden részletre nem tér ki a dokumentáció, ugyanis maga a forráskód számos kommentet (és úgynevezett summary-t) tartalmaz. A summary-k a kód egységeihez (osztályok, osztálytagok stb.) nyújtanak leírást az adott egység működéséről és használatáról.

Grafikus felület

A szoftver felülete *WinForms* technológiával, a *.NET* keretrendszerben készült el. A *Program* osztály *Main* metódusa indítja el a felületet úgy, hogy példányosítja a *MainForm* nevű (*Form*-ból származó) osztályt, majd elindítja azt.

A *MainForm* a szoftver „főablaka”; innen érhető el az összes többi funkció. A főbb funkciókhoz (blokkok szerkesztése, átnevezése, futtatása stb.) külön ablakok lettek implementálva, melyek mind a *LocalDialog* nevű osztályból származnak, melynek hierarchiája a 33. ábrán szerepel. A különböző menüpontokra kattintva (vagy billentyűkombinációkat lenyomva) ezek a dialógusok példányosodnak, majd a felhasználói interakció után (a dialógust bezárva) megsemmisülnek, és előtte bizonyos esetekben információt „adnak vissza” az interakció eredményéről a mezőiken keresztül.



33. ábra – a dialógusok hierarchiája

Minden, a szoftver által használt grafikus elem a *Control* osztályból származik, mely *WinForms* esetén a grafikus „építőelemek” ősosztálya. A felület a modellel eseményeken keresztül kommunikál, melyek gondoskodnak arról, hogy a felület konzisztens maradjon a modellel.

Témák

A szoftver lehetőséget biztosít a grafikus felület kinézetének módosítására – pontosabban a *Canvas* renderelési beállításainak és a képként való exportálás beállításainak módosítására.

A témáknak implementálnia kell az *ITheme* interfészt, mely hét stíluselemet követel meg:

- *PointedBlockEdge* – a kijelölt blokk szélén megjelenő „árnyékolt csík” kinézetét írja le.
- *Highlighting* – a lefuttatott blokk háttérszínét határozza meg (*HighlightedBlock* értéke).
- *SelectionRect* – a mutatott blokk háttérszínét határozza meg.
- *CanvasBackground* – a *Canvas* objektum hátterét adja meg.
- *StructogramBackground* – a struktogram háttérszínét írja le.
- *StructogramText* – a struktogram szövegtulajdonságait határozza meg.
- *StructogramBorder* – a struktogram szegélyeinek tulajdonságait írja le.

Jelenleg négy téma van meghatározva:

- *DefaultTheme* – az alapértelmezett téma.
- *BWTheme* – egy fekete-fehér téma.
- *DarkTheme* – egy szürkeárnyalatos téma.
- *BlueTheme* – egy „kékárnyalatos” téma.

Az *ITheme* interfészből való származtatással további témák is definiálhatók. Ekkor ügyelni kell arra, hogy a *SettingsDialog* dialógusban is megjelenjen, illetve a *SettingsObject* és a *SerializableSettingsObject* osztályban is meg kell valósítani a mentés/betöltés módját a konstruktorban. A *Canvas*ok témáját a *LoadTheme()* metódussal lehet módosítani.

Megjegyzés: jelenleg a szoftverben a témák csak a színösszeállítás módosítására használatosak, noha az ITheme adattagjai további szempontokat is meg tudnak határozni, mint például vonalvastagság, betűvastagság, szegélystílus stb.

Canvas

A grafikus felület legfontosabb része a vászon, ahol a struktogramok jelennek meg. Ezt a *Canvas* nevű osztály teszi lehetővé, mely a *UserControl*ból származik. Az osztály kódja nem rövid, hisz tartalmazza nemcsak a rajzolásért felelős kódot, hanem a felhasználói bemenet kezeléséért felelős kódot is, és emellett még a belső állapotát is fent kell tartani biztonsági feltételekkel.

A következő oldalon látható az osztály diagramja, mely a teljesség igénye nélkül a fontosabb osztálytagokat szemlélteti. Most ezen osztálytagok leírása következik.

34. ábra – a Canvas felépítése

A *Canvas* (34. ábra) röviden összefoglalva egy (felhasználótól érkező) eseményekkel befolyásolható állapotgép, mely az *UserControl.OnPaint()* eljárás felülírásával renderel annak *Graphics* objektumán keresztül a képernyőre. Három fő állapota lehet:

- *Normal* – alaphelyzet lehetséges a blokkok kijelölése kurzorral.
- *Dropping* – új blokk beszúrása van folyamatban. Blokkok ekkor nem jelölhetők ki.
- *MovingOut* – egy már meglévő blokkot mozgat a felhasználó.

Ezek a *CanvasMode* enumerációban vannak definiálva.

Az alábbiak a publikus tulajdonságok:

- *CanClick* – meghatározza, hogy a blokkokra való kattintás kiváltja-e a *BlockClicked* (és *BlockDoubleClicked*) eseményt. A dialógusokban ez ki van kapcsolva.
- *CanZoom* – meghatározza, hogy lehet-e nagyítani és kicsinyíteni a struktogramot.
- *CanSelectBlock* – meghatározza, hogy ki lehet-e jelölni blokkot. A dialógusokban ez ki van kapcsolva.
- *ZoomPanelVisible* – a nagyításért felelős sávot (alsó sáv) kapcsolja ki- vagy be.
- *ParamsFieldVisible* – a függvényt kezelő felső sávot kapcsolja ki- vagy be.
- *HighQualityRendering* – a renderelés minőségét állítja. Ha igaz, jobb minőségű lesz a kép, mintha hamis lenne.
- *HighlightedBlock* – beállítja, hogy mely' blokk legyen kijelölve. **Fontos:** ez nem ugyanaz, mint a felhasználói kijelölés. A *ProgramRunner* ezt a tulajdonságot használja a lépések követéséhez. *Null* esetén nincs kijelölt blokk.

A diagram legfelső három értéke konstans, melyek rendre az alapértelmezett betűméretet, betűtípust és húzási érzékenységet határozzák meg.

A húzási érzékenység az a távolság, amin belül a blokkmozgatási szándék még nem minősül vonszolásnak, tehát a húzott blokk még nem kerül ki a szintaxisfából.

A *_blockRects* változó a struktogramon belül lévő blokkok pozícióját és méretét *(Rectangle)* tárolják, hogy ne kelljen azokat állandóan újra és újra kiszámolni. Magának a struktogramnak a *Rectangle*-je külön van eltárolva, a *_mainBlockRectangle* változóban.

A következők a publikus osztályfüggvények:

- *CreateScreenshot()* – *Bitmap*ra rajzolja az aktuális struktogramot a beállított témában.
- *LoadThene()* – megváltoztatja *Canvas*ra vonatkozóan a beállított témát. Ha a paraméter *null*, a *DefaultTheme* kerül példányosításra.
- *ResetView()* – alaphelyzetbe állítja a nézetet és módot.
- *AlignWidth()* – a *DisplayRectangle* szélességéhez igazítja a struktogramot a nagyítás (betűméret) változtatásával.
- *AlightHeight()* – az előzővel analóg módon működik, de ez a magassághoz igazít.
- *AlignBoth()* – az előzőkhöz hasonlóan működik, de a szélességet és magasságot is egyaránt figyelembe veszi. A *ProgramRunner* egyik gombja ezt hívja meg.

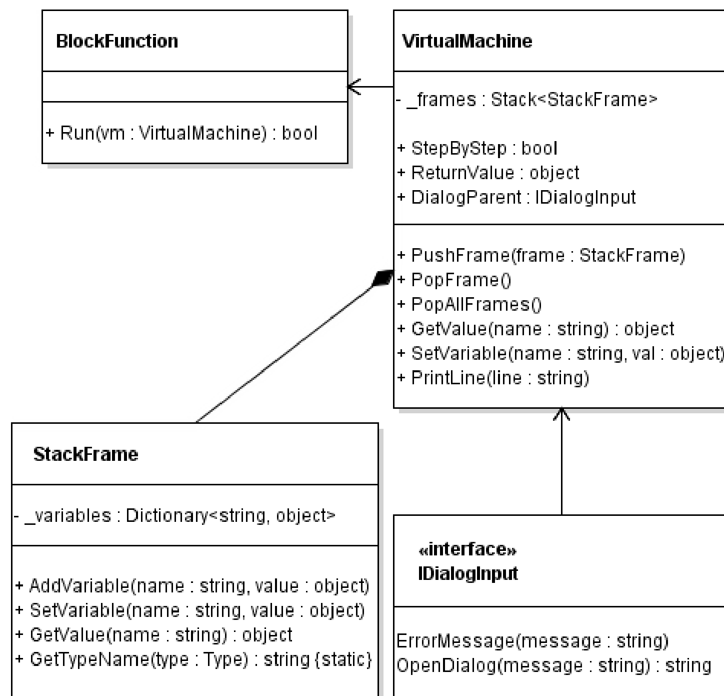
A privát adattagok részletezése is fontos, ugyanis ezeknek köszönhető a *Canvas* működése. Ez a rész a callback-függvények működésére azonban nem tér ki.

- *RemoveNode()* – törli a megadott elemet a struktogram fájából, és frissíti a *Canvas*t.
- *InsertNode()* – beszúrja az első megadott blokkba a második megadott blokkot, és szintúgy frissíti a Canvast. A szülőblokkon belüli pozíciót a harmadik paraméter határozza meg.
- *InsertBranch()* – hasonlóan működik az előzőhöz képest, azonban ez a megadott *BranchBlock*ot a másik megadott *BranchBlock* mellé szúrja be.
- *IncrementFontSize()* – megnöveli eggyel a betűméretet, és frissíti a *Canvas*t.
- *DecrementFontSize()* – lecsökkenti eggyel a betűméretet, és frissíti a *Canvas*t.
- *GetParent()* – visszaadja a megadott blokk szülőjét, és egy kimenő *(out)* paraméteren keresztül azt is, hogy hol van a blokk a szülőn belül. Szülő hiányában *null*t ad vissza.
- *GetHoveredBlock()* – megadja, hogy az adott *Point* alatt (általában a kurzor helyzete) melyik blokk található, és annak melyik részén van a megadott *Point*. Itt használja a program a *PointedPart* enumerációt.

Virtuális gép

A szoftver egyik legfontosabb funkciója, hogy képes struktogramokat végrehajtani. Ennek elengedhetetlen eszköze a virtuális gép. Habár a *BlockCode* egy elég egyszerű virtuális gépet használ, ami talán sokkal inkább mondható kontextusnak, mint virtuális gépnek. A kontextus egy objektum (*VirtualMachine* típusú), melynek egy-egy példányát használva hajtja végre a struktogramot az annak szintaxisfáját reprezentáló objektum (pontosabban a szintaxisfa *Run()* függvénye). Hogy ez pontosan hogyan történik, arra a *StructoBlock* alfejezet tér ki.

Mint az a 35. ábrán is látható, a *VirtualMachine* osztály tagja egy privát verem *(Stack)*, melyben *StackFrame* típusú objektumok vannak. Ez felel meg kvázi a függvényhívási veremnek. Minden függvényhívás előtt bekerül egy *StackFrame* a verembe a *PushFrame()* metódus által, mely tartalmazza a függvény argumentumait kiértékelve, *object* típusba csomagolva. (Ha paramétere *null,* új *StackFrame* jön létre.)



35. ábra – a virtuális gép felépítése és működése

A függvény végrehajtása során a lokális változók is a *StackFrame*-be kerülnek. A visszatérés után a függvényhez tartozó frame lekerül a veremről, és megsemmisül. Ezt a *PopFrame()* eljárás hajtja végre.

Hiba esetén (a hibakezelés lehetőségének hiányában) a teljes verem kiürül. Erről a *PopAllFrames()* gondoskodik.

A *PrintLine()* metódus az alapértelmezett kimenetre való írást teszi lehetővé. Akárhányszor meghívja a `print()` metódust, a végrehajtás a *PrintLine()*-on keresztül kiváltja a *LinePrintedEvent* eseményt, mely az arra a feliratkozókhoz eljuttatja a kiírt sort.

A *DialogParent* tulajdonság pedig a felhasználói bemenetért felelős. Az *IDialogInput* interfészt megvalósító osztályok képesek bemenetet kérni a felhasználótól, és képesek hibát dobni. Erre lényegében a tesztkörnyezetben való „felhasználói bemenet" miatt volt szükség, hiszen ott bemenetet a felhasználótól nem lehet kérni. Erről a dokumentáció végén olvashat.

A *StackFrame* osztály működése igen egyszerű. Belül egy szótárban a változó- és paraméternevekhez társítja azok értékét. Ez persze közvetlenül nem érhető el kívülről. Az *AddVariable()* metódussal új bejegyzést lehet készíteni a szótárban, a *SetVariable()*-lel meglévő bejegyzést lehet módosítani. Bejegyzés törlésére nincs szükség, így arra metódus sincsen.

Megjegyzés: mivel egy adott függvény lokális változói mind ugyanabban a StackFrame-ben, ugyanabban a szótárban vannak, így minden deklarált változó hatóköre az egész függvény.

Programnyelv

Eddig túlnyomórészt a *BlockCode* szoftver működéséről (felület, futtatás, felépítés stb.) volt szó, így itt az ideje áttérni a másik nagy „egységére” a projektnek: a struktogramok mögötti programozási nyelvre. Habár a programozási nyelv (mely a *BlockScript* nevet kapta) részletezésre került a felhasználói dokumentációban, az csak felszínesen, annak használati szempontjából érinti a nyelv „természetét”. Ez a fejezet a nyelv formális oldalát vizsgálja, és annak technikai részleteit. Lényegében a „solution” három projektje közül a *Language* projektről lesz szó.

A nyelv öt fő adattípust támogat: szám, logikai érték, string, tömb és rekord. A számokat lebegőpontos számként kezeli a szoftver, és *double* típussal reprezentálja. A logikai értékeket *bool* típussal, a stringeket *BlockString* formájában kezeli. A tömböket *List<object>* típusú vektorokként, a rekordokat pedig *Record* típusú objektumokként.

A *BlockString* osztályra azért van szükség, mert a nyelv kétféle stringet enged meg: sima és formázó stringet. (A formázó stringek kezdődnek dollárjellel.) Ezt a (logikai) információt is tárolja a *BlockString* osztály, ami afféle „wrapperként” is funkcionál a *string* típusú adattagja körül.

Tervezési elvek

A tervezés elsődleges szempontja az volt, hogy a nyelv kellően *egyszerű* legyen mind használhatóság és átláthatóság, mind az interpreter egyszerű implementálhatósága szempontjából. Továbbá a nyelvnek alkalmasnak kell lennie egy vagy több struktogram, rekorddefiníciók és konstansdefiníciók szerializálására.

Az elterjedt programozási nyelvekkel ellentétben a struktogramnak *mint „programozási nyelvnek”* nincs pontos definíciója. (Ez a szintaktikai egységesség hiányára és a típusok nem egységes definíciójára vonatkozik.) Ebből adódóan nem lehet egyértelműen, egy konkrét programozási nyelvvel reprezentálni, azonban a struktogramokról tehetünk bizonyos megfigyeléseket, melyek alapján össze lehet állítani egy irányvonalat, ami mentén a *BlockScript* programozási nyelv megvalósítható.

- Legyen Turing-teljes, és az alapvető – a struktogramokban is jelen lévő – programozási szerkezeteket támogassa.
- Ne legyen szükség a változók típusának konkrét megadására, és ad-hoc módon definiálhatóak legyenek külön deklaráció nélkül.
- Lehetséges legyen a megtervezett programot több, kisebb egységre (több struktogramra) bontani. Ez lényegében a procedurális megközelítés szükségességét, a függvények jelenlétét írja elő.
- Az alapvető matematikai műveleteket támogassa, és képes legyen logikai kifejezésekkel dolgozni. Opcionálisan stringekkel is.
- Lehessen benne összetett adatstruktúrákat használni (azaz rekordokat).

Ezen – kissé általános – feltételeket „önkényes” döntésekkel lehet tovább pontosítani, melyek alapján a programozási nyelv megalkotható.

Formális nyelvtan

A nyelv szintaxisa a BASIC programozási nyelv szintaxisát veszi alapul, elsősorban annak egyszerűsége és olvashatósága miatt. A szintaxis leírható BNF-fel, mely az alábbiakban kerül részletezésre. A nagybetűvel írt szavak a nemterminálisok, a kisbetűvel írtak a terminálisok. A vastagon szedett terminálisok a kulcsszavak; ezek szó szerint szerepelnek a kódban.

A csillag jel azt jelenti, hogy az előtte álló kifejezés nullaszor vagy többször fordulhat elő. A kérdőjel a nullaszori vagy egyszeri előfordulást jelenti, míg a pluszjel azt jelenti, hogy az előtte álló kifejezésnek legalább egyszer kell előfordulnia. A függőleges vonal opciókat választ el.

START → pre_dir* (CONST_DEF | RECORD_DEF | FUNC_DEF)* FUNC_DEF

CONST_DEF → **const** name op_assign EXPR semicolon

RECORD_DEF → **record** name op_bra name (comma name)* cl_bra

FUNC_DEF → **function** name op_bra (name (comma name)*)? cl_bra
SEQENCE
end function

SEQENCE → SKIP | (IF_ELSE | BRANCHES | WHILE | ASSIGN | CALL | RETURN)*

IF_ELSE → **if** EXPR **then**
SEQENCE
(**else** SEQENCE)?
end if

BRANCHES → **if** EXPR **then** SEQENCE
(**else if** EXPR **then** SEQENCE)+
(**else** SEQENCE)?
end if

WHILE → **while** EXPR **do** SEQENCE **next**

ASSIGN → EXPR op_assign EXPR semicolon

CALL → CALL_EXPR semicolon

SKIP → **skip** semicolon

RETURN → **return** semicolon

EXPR → number | string | name | **true** | **false** |
| **new** name | (un_op EXPR)
| (EXPR bin_op EXPR) | (EXPR op_sq_bra EXPR cl_sq_bra)
| (op_c_bra (EXPR (comma EXPR)*)? cl_c_bra)
| (op_bra EXPR cl_bra)
| CALL_EXPR

CALL_EXPR → name op_f_bra (EXPR (comma EXPR)*)? cl_f_bra

A BNF-ben a sortörések, tabulátorok, szóközök (az úgynevezett whitespace karakterek) nincsenek feltüntetve, ugyanis azoknak nincs funkcionális szerepe; az interpreter figyelmen kívül hagyja őket.

A terminálisokat reguláris kifejezésekkel lehet leírni, melyek lentebb láthatók. A {char} jelentése „kis- és nagybetűk, aláhúzásjel”. Az {endl} a sorvége karaktert jelöli, míg a {whsp} a szóköznek mondható karaktereket.

<table>
<tr><th>Terminális</th><th>Reguláris kifejezés</th></tr>
<tr><td>name</td><td>{char} [{char}0-9]*</td></tr>
<tr><td>op_assign</td><td>:=</td></tr>
<tr><td>comma és semicolon</td><td>, és ;</td></tr>
<tr><td>op_bra és cl_bra</td><td>\(és \)</td></tr>
<tr><td>op_sq_bra és cl_sq_bra</td><td>\[és \]</td></tr>
<tr><td>op_c_bra és cl_c_bra</td><td>\{ és \}</td></tr>
<tr><td>un_op</td><td>not | -</td></tr>
<tr><td>bin_op</td><td>\+ | - | * | / | <=? | >=? | !?= | \. | \^ |
mod | div | in | and | or | xor | imp</td></tr>
<tr><td>number</td><td>-? (inf | NaN | epsilon | (0 | [1-9][0-9]*) (\.[0-9]+)?)</td></tr>
<tr><td>string</td><td>\$?\”[^\”]*\”</td></tr>
<tr><td>pre_dir</td><td># {char}+ {whsp}+ [^{endl}]* {endl}</td></tr>
</table>

Az *op_f_bra* és *cl_f_bra* tokeneket a lexer kontextus alapján az *op_bra* és *cl_bra* tokenek helyett adhatja vissza.

Mint az már említésre került, a szoftver támogatja a magyar kulcsszókészlet használatát, illetve egyénit is lehet definiálni. Így a fentebb leírt (eredeti, angol nyelvű) kulcsszavak és operátorok a beállított készlettől függően eltérhetnek.

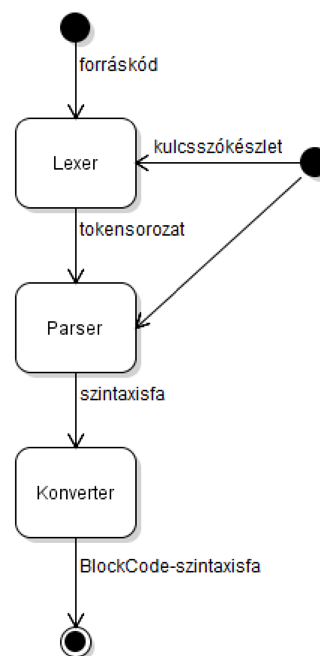
Kommentet is lehet írni a kódba. Kommentnek számít a // jelektől a legelső sortörésig tartó karaktersorozat. A lexer ezeket nem adja vissza. A preprocesszor-direktívákat nem a parser értelmezi; azok a fordítás előtt értelmeződnek, és a fordító a direktíváktól (és kommentektől és sortörésektől) mentes bemenetet kapja meg.

A fordító a nyelv értelmezéséhez egy rekurzívan megírt LL(1)-es parsert és egy speciális shunting-yard algoritmust használ, noha a leírt nyelv nem LL(1)-es. Ennek a parsernek (és a hozzá tartozó lexernek) a részletezése következik.

Az interpreter felépítése

A *BlockScript* nyelv értelmezésére írt parser (mely a *Language* projekt alatt található) a szabványosnak mondható elven működik; a megszokott komponensekből áll, melyek a 36. ábrán láthatók.

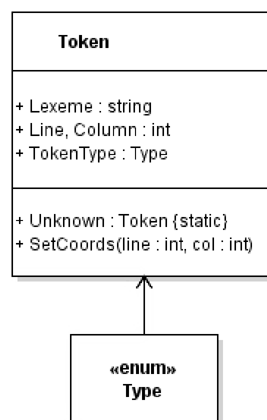
0. A felhasználó megnyit egy kódfájlt, mely egy string-változóba olvasódik be.
1. A beolvasott kód (egy tetszőleges *IKeywordSet* típusú kulcsszókészlettel együtt) átadódik a lexernek, mely egy *IEnumerable<Token>* típusú „tokenfelsorolót" ad vissza.
2. A parser megkapja a tokeneket és a kulcsszókészletet, majd egy *SeqBlockScriptNode* típusú objektumban visszaadja a függvényekből, rekordokból és konstansokból álló szintaxisfát.
3. Mivel a *Language* projekt a modularitás szellemében íródott, önálló szintaxisfa-struktúrája van, így azt át kell alakítani a *BlockCode* saját reprezentációjára. A transzformációt a *BlockCode* projektben a *BlockExtensions.ToMainSequence()* függvény végzi.



36. ábra – a parse-olás menete

Lexikális elemző

A „nulladik" lépés, a programkód beolvasása után a beolvasott karakterek tokenizálása következik. A tokenizálás a *.NET* saját reguláris mintaillesztőjével működik, amit a *System.Text.RegularExpressions* névtér eszközei tesznek lehetővé. A tokentípusokhoz reguláris kifejezések vannak rendelve, melyek alapján a *Regex* osztály képes a kódot tokenenként feldolgozni. Ez lényegében nemdeterminisztikus automatákat hoz létre reguláris kifejezésekből. A tokeneket reprezentáló osztály a *Token* struktúra, mely a 37. ábrán van részletezve.



37. ábra – a tokenek

Előfordul, hogy egy adott karaktersorozatról nem lehet egyértelműen eldönteni csupán reguláris elemzéssel, hogy mi a típusa. Például az **a - b** kifejezésben a '-' token típusa a kétoperandusú kivonásjel, míg a **-b** kifejezésben ugyanez a token az egyoperandusú „negatív előjel". Ezeknek a kétértelműségeknek meghatározására a lexer figyelembe veszi a token előtt és után álló tokeneket – vagy éppen azok hiányát – és azok alapján határozza meg, hogy melyik típusúnak tekintse azt.

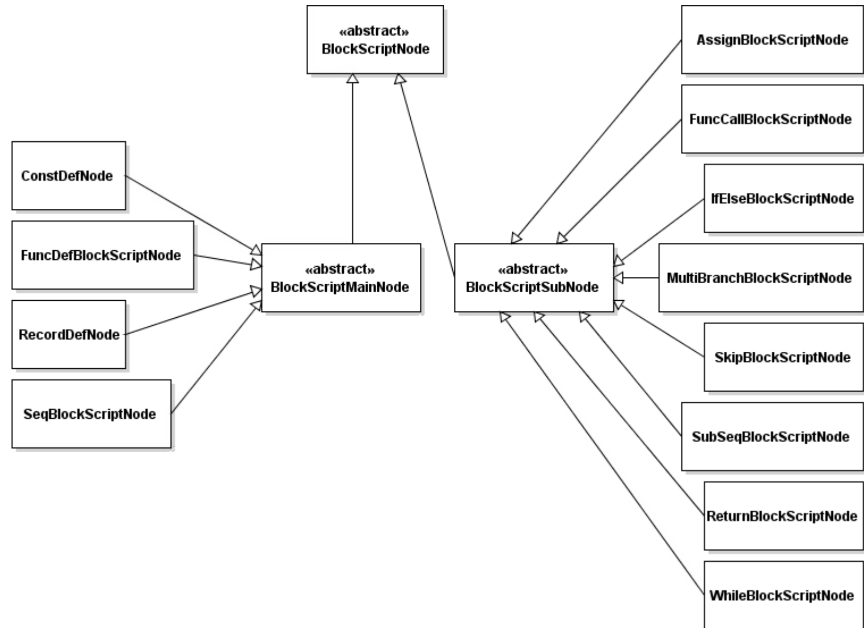
Ugyanez igaz a névtokenekre, melyek vagy változót/konstanst vagy függvényhívást jelölnek; illetve a „sima" zárójelre is, amely lehet függvényhívásé is. Utóbbi esetben a nyitózárójel zárójelét verem segítségével határozza meg a lexer.

Megjegyzés: a változó/konstans és függvényhívás közti különbséget nem feltétlenül szükséges felismerni, azonban segíti a parse-olást, és az esetleges bővítésekhez is hasznos lehet.

Mint az már említésre került, a *Language* projekt más reprezentációt használ, mint a *BlockCode*. Ez az alfejezet a *Language* reprezentációját tárgyalja.

A 38. ábrán, lent látható a szintaxisfát felépítő osztályok hierarchiája. A teljesség igénye nélkül az ábrán csak az öröklődések láthatók.

Minden osztály/csúcs az absztrakt *BlockScriptNode* objektumból származik. Ennek 12 konkrét leszármazottja van, melyek két kategóriába sorolhatók aszerint, hogy hol fordulhatnak elő a kódfájlban.



38. ábra – a nemterminálisok reprezentációja

A kódfájl, maga függvényeket, konstansokat és rekordokat tartalmazhat csak. Ezen csúcsai a fának a *BlockScriptMainNode* osztályból származnak. A függvényeken belül előforduló csúcsok pedig a *BlockScriptSubNode* osztályból származnak.

Megjegyzés: az „IfElse” és a „MultiBranch” azért különböző osztályok, mert eredetileg a struktogramokban csak „IfElse” szerepelhetett.

Az egyes osztályok (csúcsok) szerepe alább olvasható:

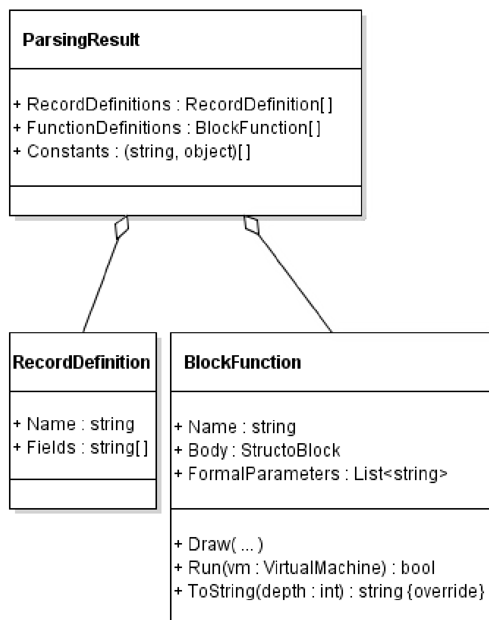
- *ConstDefNode* – konstansdefiníciót reprezentál.
- *FuncDefBlockScriptNode* – függvénydefiníciót reprezentál.
- *RecordDefNode* – rekorddefiníciót reprezentál.
- *SeqBlockScriptNode* – az előbbi három blokk szekvenciáját reprezentálja, és egyben ennek az objektumnak példányaként adja vissza a parser a szintaxisfát. Implementálja az *IEnumerable<BlockScriptMainNode>* interfészt. Akárhány gyereke lehet.
- *AssignBlockScriptNode* – az értékadásokat reprezentálja. A csúcsnak két *Expression* típusú paramétere van: *AssignTo* és *Expression*. Az előbbi az értékadás bal oldala, a második értelemszerűen a jobb oldala.
- *FuncCallBlockScriptNode* – a „nem-kifejezésbeli” függvényhívásokat reprezentálja. Két paramétere a *string* típusú *FunctionName* és a *List<Expression>* típusú *Parameters*. Az első a hívott függvény neve. A második a hívásban meghatározott paramétereket tárolja.
- *IfElseBlockScriptNode* – a kétágú elágazást reprezentálja egy *Condition* nevű *Expression* típusú mezővel, illetve a csúcs két gyerekével: *TrueBody* és *FalseBody*. Ezek rendre az igaz (bal oldali) és hamis (jobb oldali) ágra hivatkoznak.

- *MultiBranchBlockScriptNode* – többágú elágazás. Tetszőlegesen sok gyereke van, melyek egy-egy ágat és a hozzájuk tartozó feltételt tárolják egy *List<(Expression, BlockScriptSubNode)>* típusú, rendezett párokat tároló vektorban.
- *SkipBlockScriptNode* – egy `skip;` utasítást reprezentál. Nincs gyereke, se paramétere.
- *SubSeqBlockScriptNode* – *BlockScriptSubNode* típusú csúcsok szekvenciája. Megvalósítja az *IEnumerable<BlockScriptSubNode>* interfészt. Ez abban különbözik a *SeqBlockScriptNode* csúcstól, hogy míg az a nyelv legfelső szintjét képviseli („fő" nemterminális), addig ez függvényeken belül, és függvényeken belüli csúcsokban lévő szekvenciákat reprezentálja. Ennek is akárhány gyereke lehet.
- *ReturnBlockScriptNode* – a `return;` utasítást reprezentálja. Egy mezője van: az *Expresison* típusú *Value*, mely a visszatérési értéket írja le. *Null* is lehet az értéke; az az érték nélküli visszatérést jelenti.
- *WhileBlockScriptNode* – a WHILE ciklusokat reprezentálja egy feltétellel (*Expression* típusú *Condition* mező) és egy gyerekével, a *BlockScriptSubNode* típusú *Body*val.

Minden *BlockScriptNode* megvalósítja az *IEquatable<BlockScriptNode>* interfészt, előírja az *Equals()* függvényt. Ennek a tesztelés során van igazán jelentősége, amikor szintaxisfák csúcsait kell összehasonlítani.

A parser, mint már említésre került, a „rekurzív leszállás" elvén készült. Minden nemterminálisnak (kivéve a *IfElseBlockScriptNode*-nak és a *MultiBranchBlockScriptNode*-nak) meg van feleltetve egy függvény, amely vagy egy *BlockScriptNode*-ot ad vissza, vagy *null* értéket. *(A null érték azt jelenti, hogy nem sikerült az illesztés.)* Az előbbi két nemterminális azért kivétel, mert azokat ugyanaz a függvény implementálja.

Konverter



39. ábra – a parse-olás eredménye

A szintaxisfa-reprezentációk közötti leképezést (egyirányú, *Language*-reprezentáció → *BlockCode*-reprezentáció) a *BlockExtensions.ToMainSequence()* függvény végzi, ami már említésre is került. Ez az átadott szintaxisfát bejárja, és azt egy vagy több új szintaxisfává tördeli, illetve kinyeri belőle a rekord- és konstansdefiníciókat, amiket végül egy *ParsingResult* típusú objektumban ad vissza. Szerkezetük a 39. ábrán látható.

A *ParsingResult* objektumok három részre bontva tartalmazzák a kódfájlból beolvasottakat: rekordok definíciói *RecordDefinition*-tömb formájában, függvénydefiníciók *BlockFunction*-tömb formájában, illetve konstansdefiníciók *(string, object)* rendezett párokból álló tömbként.

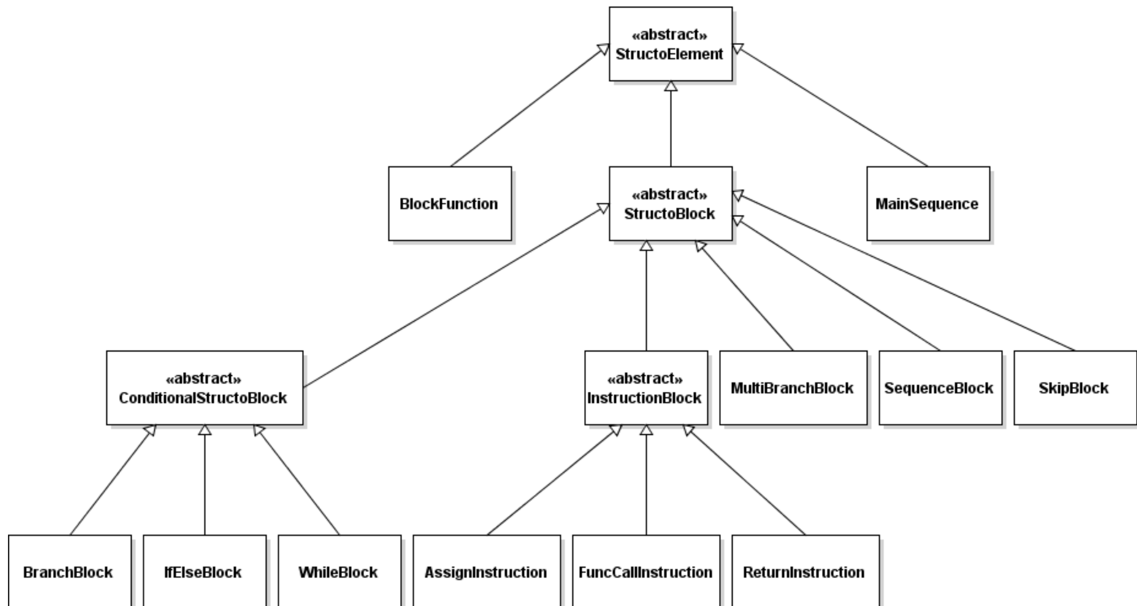
Megjegyzés: a konstansok értéke – akárcsak a változók értéke – object típusú objektumokba van csomagolva; azokként adódnak át. Ezt érdemes lehet egy erősen típusos osztályhierarchia példányaiként reprezentálni a típushelyesség kedvért.

Szintaxisfa és kifejezések

Az előbbiek lefedték a *Language* projekt lényegét, annak működését. Ez a fejezet visszatér a *BlockCode* működéséhez, és annak nyelvi reprezentációját tárgyalja.

Az algoritmusok reprezentációja

Az alábbi (40.) ábrán látható a struktogramok reprezentációjához használt osztályok hierarchiája:



40. ábra – a blokkok hierarchiája

Ez az osztályhierarchia teszi lehetővé a struktogramokkal kvázi *ekvivalens* nem-absztrakt szintaxisfák építését, melyek rögtön futtathatók is. Értelemszerűen a futtatás az említett szintaxisfa fentről-lefelé történő kiértékelését jelenti, melynek pontos módja a következőkben kerül leírásra.

Megfigyelhető, hogy míg a *Langauge* projekt a rekordokat *(RecordDefNode)* és a konstansokat *(ConstDefNode)* is a szintaxisfa részének tekinti, addig a *BlockCode* külön kezeli őket. Erről már volt is szó. A következőkben a fent látható osztályok kerülnek részletezésre.

StructoBlock

A *StructoBlock* osztályból (41. ábra) öröklődik minden olyan osztály, mely egy struktogram blokkja lehet.

«abstract»
StructoBlock

HORIZ_PADDING, VERT_PADDING : float {const}

+ GetWidth(f : Font) : int
+ GetHeight(f : Font) : int
+ GetChildrenAndSelf() : IEnumerable<StructoBlock>

+ Draw(...)
+ Execute(vm : VirtualMachine) : bool

41. ábra – a blokkok ősosztálya

Talán legfontosabb eljárása a *Draw()* absztrakt metódus. Ez végzi az adott blokk (és esetleg annak gyerekeinek rekurzív) renderelését egy tetszőleges objektumra, amelyre lehet kérni egy *Graphics* objektumot. Ilyen például a *Canvas*. Négy paramétere van, melyek hely hiányában a jobb oldali diagramon nem lettek feltüntetve, de a forráskódban részletezésre kerültek. Érdemes azonban itt részletezni a *DrawingStyle* típusú paramétert. A *DrawingStyle* osztály adja át az eljárásnak a rajzoláshoz szükséges információkat a *Font*, *BorderPen* és *TextBrush* mezők használatával. A negyedik paraméter egy szótár *(Dictionary<StructoBlock, Rectangle>)*, mely egy afféle gyűjtőobjektumként funkcionál. Ha értéke nem *null*, a struktogram egyes blokkjainak *Rectangle*-je (pozíciója és mérete) belekerül. Ez a *Canvas*nak fontos, ugyanis ezen *Rectangle*-ök alapján dönti el, hogy az egyes blokkok hol vannak, és melyeken áll a kurzor éppen.

Az *Execute()* osztályfüggvény a hozzá tartozó blokkot végrehajtja a függvénynek paraméterként átadott *VirtualMachine* objektumon. Ez az osztály már részletezésre került a *Virtuális gép* fejezetben. A függvény visszatérési értéke hamis, ha az adott blokk a végrehajtás végét eredményezte (például egy `return;` utasítás elérésével). Különben igaz.

1. Ez a függvény mindig azzal kezd, hogy megnézi, fel van-e iratkozva az adott objektum „végrehajtás megkezdve" eseményére valamilyen callback-függvény. Ha igen, meghívja az(oka)t.
2. Majd ha a virtuális gép *StepByStep* tulajdonsága igaz, annyi milliszekundumot alszik, amennyit a felhasználó megadott.
3. Végrehajtja az adott blokkhoz tartozó vérehajtási kódot. Ez persze minden blokktípus esetén más, így ezek a következő fejezetben kerülnek leírásra.
4. Ha a végrehajtás nem szakadt meg, meghívódnak a „végrehajtás befejezve" esemény callback-függvényei, ha vannak. *Például a ProgramRunner ezt (és az első pontban leírt eseményt) használja a követésre és a lépésenkénti végrehajtásra.*

A *GetChildrenAndSelf()* egy felsorolót ad vissza, mellyel posztorder módon lehet bejárni a szintaxisfa csúcsait. Ezt is a *Canvas* használja annak megállapítására, hogy a kurzor éppen melyik blokk fölött áll.

A *GetWidth()* és *GetHeight()* az adott blokk szélességét, illetve magasságát adja vissza a megadott *Font* függvényében. Nyilván ezek a függvények is bejárják a szintaxisfát. *Ez jó lehetőség némi optimalizálásra.*

Hogy az egyes osztályok miként határozzák meg méretüket, az az egyes blokkok alfejezetében kerül definiálásra.

A *HORIZ_PADDING* és *VERT_PADDING float* típusú konstansszorzók, melyek meghatározzák, hogy a blokkokban lévő szöveg méretéhez hogyan viszonyul az egész blokk mérete.

ConditionalStructBlock

Az ebből az absztrakt osztályból származó alosztályok mind rendelkeznek az *Expression* típusú *Condition* tulajdonsággal. Mint azt a név is sugallja, ez az osztály a feltétellel rendelkező blokkokat foglalja egybe. Ezek az alábbiak:

BranchBlock

Egy többágú elágazás *(MultiBranchBlock)* ágait reprezentálja. Megfelelője a *Language* projektben a *MultiBranchBlockScriptNode* osztály. Csak *MultiBranchBlock*on belül fordulhat elő. Ezt az invariáns tulajdonságot a szerkesztés során a *Canvas* eljárásai biztosítják is.

Végrehajtása során kiértékeli a feltételét, majd, ha az igaz, végrehajtja a *Body* mezőben megadott elágazásmagot. Ennek az osztálynak *Execute()* függvényét csakis a *MultiBranchBlock* osztályból szabad meghívni.

Magassága a feltétel magasságágának és a mag magasságának összege. Szélessége az elágazás magjának szélességével egyenlő.

IfElseBlock

A kétágú elágazásokat reprezentálja az *IfElseBlock* osztály. Végrehajtása annyiban tér el a *BranchBlock* végrehajtásától, hogy ha a feltétel igaz, a *TrueBranch* fut le. Különben a *FalseBranch*, tehát mindenképpen valamelyik.

Magassága megegyezik a magasabb ág magasságának és a feltétel magasságának összegével. Szélessége mindkét ág szélességének összege.

WhileBlock

A *WhileBlock* a WHILE ciklusnak megfelelő osztály. Kiértékeli a feltételt (*Condition*), majd ha az igaz, lefut a *Body* mezőben megadott fa. Ezt addig ismétli, amíg a feltétel igaz, és a mag a lefutása után igaz értékkel tér vissza (tehát nem futott hibába vagy `return;` utasításba).

Magassága egyenlő a ciklusfeltétel magasságának és a ciklusmag magasságának összegével. Szélessége a bal oldali „sáv” és a ciklusmag szélességének összege, ahol a „sáv” szélessége a beállított betűmérettel egyenesen arányos.

InstructionBlock

Utasítást reprezentáló absztrakt blokk. A szélességek és magasságok számításában ez az egyik elemi osztály, ugyanis méretei tisztán az értékadás „szövegének” szélességével és magasságával ekvivalensek, így az egyes öröklődő osztályok nem is írják felül a *GetWidth()* és *GetHeight()* függvényeket. Egy *InstructionBlock* szélessége maximum **768 pixel** lehet.

AssignInstruction

Az értékadást reprezentálja. Az *AssignTo* mező az értékadás bal oldala, az *Expression* pedig a jobb. Végrehajtásakor meghívja az *Assign()* függvényt a kapott *VirtualMachine*-nel, ami elvégzi az értékadást. Ez a függvény megnézi, hogy balérték-e a bal oldal. Ha nem, *ExpressionException* kivételt dob. Ha igen, akkor a bal oldal típusától függően elvégzi az értékadást. A bal oldal lehet:

- Változó (Variable)
- Indexer (Indexer)
- Bináris operátor *(BinOp)*

Más bal oldalnak nem lehet értéket adni, tehát például a `new Record.field := valami;` értékadás nem megengedett. Az értékadásokról többet a felhasználói dokumentáció *Változók és értékadás* alfejezetében talál.

FuncCallInstruction

Függvények utasításként való hívását reprezentálja. Kikeresi a *FuncName* mezőben megadott függvényt a *GlobalFunctions* tárolóiból, és végrehajtja azt az *Arguments* mezőben megadott paraméterek „evaluációja” után. A `print()` és `test()` metódusokat speciálisan kezeli.

ReturnInstruction

A `return;` utasítást reprezentálja. Végrehajtása igen egyszerű: beállítja a virtuális gépben a visszatérési értéket (ha van), majd *false*-t ad vissza a végrehajtás végét jelezve – ez az információ felfelé propagálódik a szintaxisfában, majd az első *Execute()*-hívás véget ér. A visszatérési érték a *ExprToReturn* mezőben van, ami, ha nincs visszatérési érték, *null* értékű.

MultiBranchBlock

A *MultiBranchBlock* önmagában nem jelölhető ki a *Canvas* objektumokon, csupán a *BranchBlock*ok közvetlen szülőcsúcsaként funkcionál. Nem tartalmazhat nulla ágat. Implementálja az *IEnumerable<BranchBlock>* interfészt, hogy könnyen végig lehessen lépkedni az ágakon.

Magassága egyenlő a legmagasabb ágának magasságával, és szélessége egyenlő az ágai szélességének összegével. A feltételek magassága (a blokk felső része) függ az összes feltétel magasságától. Minden feltétel magassága egyenlő a legmagasabb feltétel magasságával.

SequenceBlock

A szekvenciát (mint programozási szerkezetet) reprezentálja. Szintén láthatatlan blokk, és manuálisan nem adható hozzá struktogramokhoz. Szükség esetén a grafikus felület (*Canvas* osztály) automatikusan hozza létre, amikor a felhasználó blokkokat egymás után tesz, hiszen ekkor az egyetlen blokkot több blokk szekvenciájára kell cserélni.

Továbbá implementálja az *IEnumerable<StructoBlock>* interfészt, hogy egyszerűen lehessen az elemein iterálni.

Grafikus megjelenése nincsen; csupán az elemei jelennek meg egymás alatt, vízszintes vonallal elválasztva. Akárcsak a *MultiBranchBlock*, a *SequenceBlock* önmagában nem jelölhető ki a *Canvas* objektumokon.

Magassága megegyezik az elemei magasságainak összegével, és szélessége egyenlő az elemei közül a legszélesebb szélességével.

SkipBlock

A `skip;` utasítást reprezentálja. Újonnan létrehozott szerkezetekben (WHILE ciklus, elágazások) és függvényekben/metódusokban fordul elő afféle „helytartóként”, ugyanis a felsorolt elemek nem lehetnek „üresek”. Manuálisan nem adható hozzá struktogramokhoz, kivéve a mentési fájl manuális szerkesztésével.

Végrehajtása nem okoz állapotváltozást, de – a többi blokkhoz hasonlóan – eseményeket okoz.

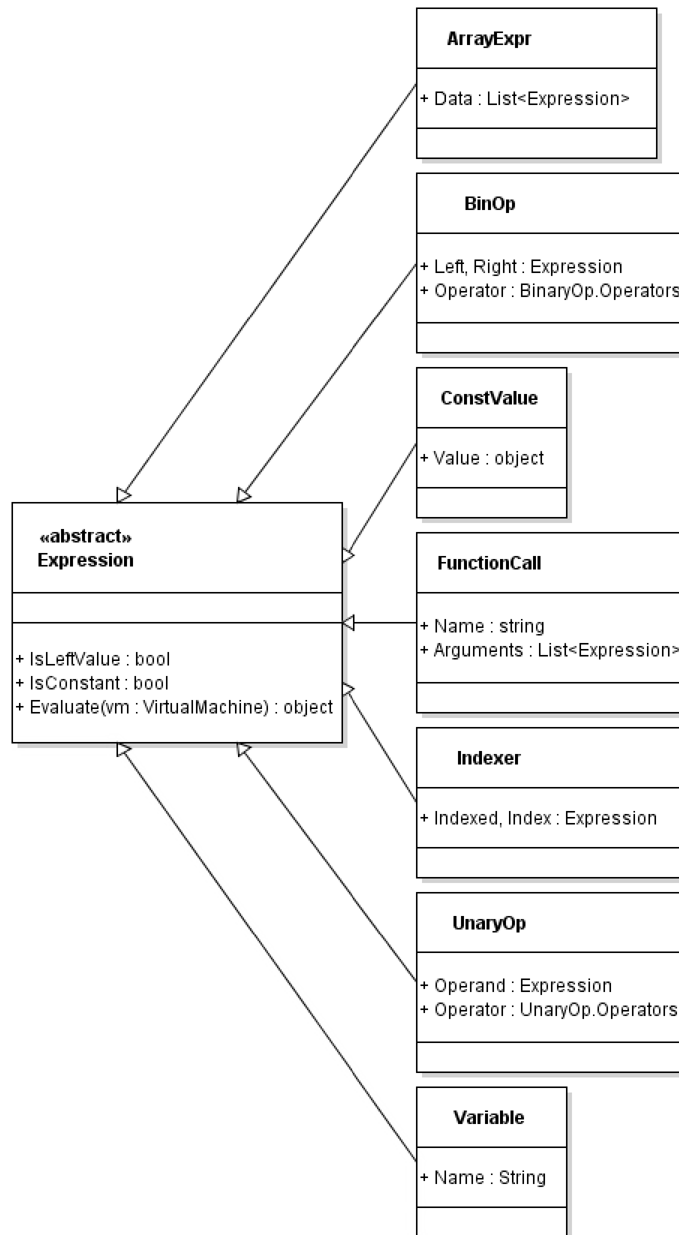
Grafikus megjelenése egy vízszintes vonal a blokk közepén.

Kifejezések reprezentációja

Programozás során akárhányszor értéket adunk, feltételt fogalmazunk meg, függvényt vagy metódust hívunk meg, elkerülhetetlen, hogy kifejezéseket fogalmazzunk meg a kívánt érték leírására. Az algoritmusok írása során használt kifejezések reprezentációja a fordítóban és a BlockCode-ban is egyaránt absztrakt szintaxisfákkal történik, akárcsak magának az algoritmusnak a reprezentációja.

A kifejezések *Expression* típusú csúcsokból állnak. Az *Expression* osztály (42. ábra) absztrakt; csupán a különböző típusú részkifejezések „ősosztálya".

A kifejezések hétféle elemből állhatnak. Legkisebb elemei a konstans termek (*ConstValue*) és a változó termek (*Variable*). A szintaxisfában ezek a levélcsúcsok. Összetett termeket a termek rekurzív egymásba ágyazásával kapunk. Erre használható a tömb (*ArrayExpr*), a bináris operátorok (*BinOp*), a függvényhívás (*FuncCall*), az indexer (*Indexer*) és az unáris operátorok (*UnaryOp*). Ezek részletezése következik.



42. ábra – a kifejezések hierarchiája

Konstans kifejezés

A konstans kifejezések az olyan kifejezések, melyek értékét kontextus nélkül, fordítási időben meg lehet határozni, azaz üres *VirtualMachine*-ben is ki lehet értékelni. Annak meghatározására, hogy egy adott kifejezés konstans-e, az *Expression.IsConstant* tulajdonság használható, mely definíciója:

1. Minden *ConstValue* konstans.
2. A *Variable* csak akkor konstans, ha konstansként regisztrálva van a név a *GlobalConstants* osztályban.
3. A *BinOp* és *UnaryOp* konstans, ha az argumentuma(ik) konstans(ok).
4. Az *ArrayExpr* konstans, ha a *Data* mező minden eleme konstans.
5. Az *Indexer* konstans, ha az *Indexed* és *Index* mező egyaránt konstans.
6. A *FunctionCall* konstans, ha az *Arguments* mező minden eleme konstans.

Balérték

Az olyan kifejezéseket, melyek egy olyan objektumot (vagy egy objektum részét) határoznak meg, melynek értékét meg lehet változtatni, balértékeknek nevezzük. Az *Expresison.IsLeftValue* annak meghatározására szükséges, hogy egy adott kifejezés balérték-e. Ennek definíciója:

1. Minden *Variable* balérték, és egy *ConstValue* sem balérték.
2. A *BinOp* balérték akkor, ha a *Left* mezője balérték, és az *Operator* mezője *BinaryOp.Operators.Dot*.
3. Az ArrayExpr, a FunctionCall és a UnaryOp sosem balérték.
4. Az *Indexer* balérték, ha az *Indexed* mező balérték.

Az *Expression*ök rendelkeznek egy *Evaluate()* tagfüggvénnyel. Ezzel a függvénnyel lehet a kifejezést kiértékelni, annak szintaxisfájának bejárásával. Bemenetként agy *VirtualMachine* objektumot kap, ami (akárcsak a *StructoBlock*ok esetén) kontextust biztosít a függvénynek, hogy a nem-konstans kifejezéseket is ki tudjon értékelni. Konstans kifejezések esetén akár egy „üres” *VirtualMachine*-nel is meg lehet hívni. A függvény absztrakt; az *Expression*ből származó osztályok különböző módon valósítják meg:

- **ArrayExpr** – a Data mező minden eleme (rekurzívan) kiértékelődik, majd a függvény egy *List<object>* típusú vektorban adja vissza őket.
- **BinOp** – kiértékelődik a bal, majd jobb oldali kifejezés, majd azok típusától függően dől el, hogy melyik bináris függvény értékét adja vissza a függvény. A lehetséges kiértékelések a felhasználói dokumentációban a *Kifejezések* alfejezetben fel lettek sorolva. Az operátorok a *Language* projektben a *BinaryOp.Operators* enumerációban vannak felsorolva. A *Dot* operátor speciálisan kezelendő.
- **ConstValue** – a kiértékelés során ez az egyik levélcsúcs. Kiértékelése egyszerűen a *Value* mező visszaadását jelenti. Fontos arra ügyelni, hogy csak a megengedett típusokkal szabad példányosítani (*double*, *bool*, *BlockString*, *List<object>*, *Record*).
- **FunctionCall** – kikeresi a *Name* mezőben megadott függvényt a *GlobalFunctions* statikus osztályból, majd az *Arguments* mező elemeit sorban kiértékeli, és a kapott értékekkel meghívja a függvényt reprezentáló *FunctionDefinition* objektum *Call()* függvényét.
- **Indexer** – kiértékeli az *Indexed* mezőt, majd ha a kapott érték indexelhető, az *Indexer* mezőt is kiértékeli, és visszaadja az *Indexed* értékének indexelésével kapott értéket.
- **UnaryOp** – kiértékeli az *Operand* mezőt, majd a kapott érték típusától függően végrehajtja az *Operator* mezővel megadott műveletet, és annak eredményét visszaadja. A *New* operátor kiemelten van kezelve.
- **Variable** – a másik „levélcsúcs”. Lekéri a kapott *VirtualMachine*-től a *Name* mezőben megadott változó értékét. Ha nem lehetséges, mert a változó nincs definiálva, megpróbálja konstansként értelmezni a nevet a *GlobalConstants* osztályt használva. A kapott értéket visszaadja.

Globális tárolók

Mint az már többször is említésre került, a gyűjtemények forráskódjában három „entitás” fordulhat elő: függvény, rekord és konstans. Ezeket az entitásokat a *BlockCode* statikus osztályokban, globálisan tárolja. Az előző fejezetben szó esett a *GlobalFunctions* és *GlobalConstants* osztályok használatáról. Most ezek és a *GlobalConstants* osztály kerül kifejtésre.

GlobalFunctions

Szoftverbeli függvényeket, és az azok kezeléséhez (törléséhez, betöltéséhez) szükséges eljárásokat foglalja egybe. Az egyes függvényeket *FunctionDefinition* objektumokkal írja le. Háromféle függvény között tesz különbséget: beépített *(built-in)*, felhasználói *(user)* és alapértelmezett könyvtárból származó *(std)*.

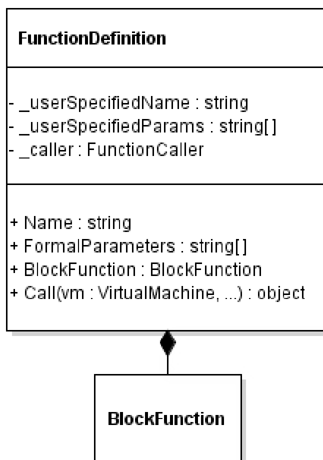
A felhasználói függvények azok, melyek a betöltött gyűjteményben találhatók. Ezek a gyűjtemény betöltésekor kerülnek a *GlobalFunctions* tárolójába, és a gyűjtemény bezárása vagy cseréje esetén törlődnek. Az ilyen függvények *BlockFunction*ökként vannak reprezentálva, amire egy-egy *FunctionDefinition* hivatkozik.

Az „standard-függvények” az alapértelmezett könyvtárból töltődnek be a tárolóba, és nem lehet őket se törölni, se felülírni (a szoftver használatával). Ezek a felhasználói függvényekhez hasonlóan működnek a *FunctionDefinition*jükön belül.

A beépített függvények némiképp máshogy működnek, hisz nem struktogramok formájában vannak definiálva. A *BuiltInFunctions* statikus osztályon belül vannak nyelvi szintű függvények formájában, így a hívásuk nem történhet úgy, ahogy a többi függvényé. Éppen ezért van szükség a *FunctionCaller* nevű delegáltra, melyre illeszkedő függvények egyetlen feladata, hogy lehetővé tegye, hogy mind a beépített, mint a felhasználói és standard függvényeket ugyanúgy lehessen meghívni. Beépített függvények esetén a *FunctionDefinition BlockFunction* tulajdonsága *null*.

Az osztálynak van egy NextUnnamedFuncName nevű tulajdonsága, mely megadja a sorban következő szabad alapértelmezett függvénynevet. Az első a „Névtelen0”, ahol a végén lévő szám növekszik minden egyes (felhasználói) függvénnyel.

FunctionDefinition



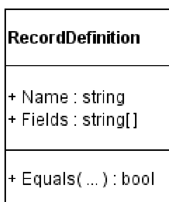
43. ábra - függvénydefiníció

A 43. ábrán szerepel a *FunctionDefinition* osztálynak a diagramja. Látható rajta a *Call()* függvény, mely egy *VirtualMachine*-példányt és egy tömbnyi *object* objektumot kap paraméterként. Az előbbi a kontextus miatt kell – és azért, hogy a `read()` függvény tudjon kommunikálni a felhasználóval. A második pedig a függvény **előre kiértékelt** paraméterei, ergo a kiértékelést a hívó végzi.

A legfelső két mező a függvény neve és formális paraméterei. Beépített függvényeknél is van név és formális paraméter (a felhasználó szemszögéből is), így ez a mező (és az előbbi csak akkor *null*, ha a *FunctionDefinition* által reprezentált függvény felhasználói vagy standard. A *_caller* mező a híváshoz lefuttatandó kódra mutat; erről az előbb szó is esett.

A felső két tulajdonság azért szükséges, hogy minden típusú függvényt egyformán lehessen használni. Beépített függvényeknél ezek a tulajdonságok a nekik rendre megfelelő osztályszintű változókat adják vissza, különben csak afféle proxyként szolgálnak a *BlockFunction* nevű tulajdonsághoz. Ez mutat a felhasználói/standard, struktogrammal definiált függvényre.

GlobalRecords



44. ábra – rekorddefiníció

A rekorddefiníciók tárolásáért és kezeléséért a szintén globális és statikus *GlobalRecords* osztály felelős. Mivel beépített rekordok nincsenek, itt csak **felhasználói** és **standard** definíciók vannak. Az osztály a tároláson, létrehozáson és törlésen kívül nem szolgál egyéb funkcióval.

RecordDefinition

A rekordok definícióját reprezentáló osztálydiagram a 44. ábrán látható. Látható, hogy a rekordok a nevükkel és mezőneveikkel vannak definiálva. Az osztály implementálja az *IEquatable<RecordDefinition>* interfészt.

GlobalConstants

A konstansok definícióiért felelős osztály még egyszerűbb. Itt is csak felhasználói és standard definíciók szerepelhetnek, melyek egy *Dictionary<string, object>* asszociatív tárolóban vannak. Ez az osztály sem szolgál az alapvető funkcióknál többel.

Érdemes megemlíteni, hogy a konstansok tényleges értékükkel vannak definiálva, nem pedig a definíciójukban megadott szintaxisfájukkal, ergo a mentési fájlba mentéskor sosem kerül kifejezés, csak valós érték.

Exportálás

A gyűjtemények exportálása összetett feladat. A problématér nagysága miatt (lehetséges szintaxisfák) nem is egzakt módon definiált. A feladat az, hogy egy adott gyűjteményhez adjuk meg az annak „lehetőleg minél jobban” megfelelő C#-kódot vagy Python-kódot.

A struktogramok természetéből adódik, hogy egy adott struktogramból (és formális paramétereiből) nem mindig lehet annyi információt kiszűrni, amennyi a fenti nyelvekre való exportáláshoz szükséges.

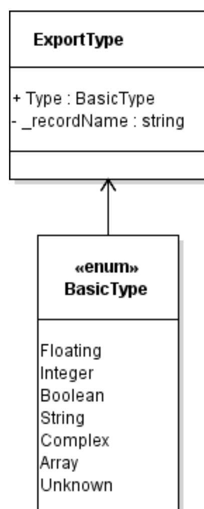
Hogy említsek egy példát, az alábbi programkód nem konvertálható egyértelműen C# nyelvre:

```
x := read();
print(len(x));
```

A `read()` visszatérési értéke, sőt, annak típusa is csak futási időben derül ki, tehát nem lehet tudni, hogy a második sort hogyan kell lefordítani. Ha `x` tömb, a hozzá tartozó C#-kód *x.Count*, míg ha string, akkor *x.Length*. Ha `x` típusa más, le se lehet fordítani. *Megjegyzés: a fenti függvényt nem is engedi az exporter exportálni.*

Bemenet

Az előbbi problémát áthidalandó van a szoftverben a *TypeInputDialog*, mely a felhasználónak kilistázza a gyűjtemény (exportálandó részhalmazának) változóit és formális paramétereit, és azok típusának pontos meghatározását a felhasználótól várja. Persze előtte azért megpróbál annyi információt kideríteni a változókról és paraméterekről, amennyit csak tud. Erre főként a *Traversals.FirstAssigment* függvényt használja, mely megadja egy változóhoz vagy paraméterhez az első értékadását. Ha az értékadás jobb oldalán szereplő kifejezés típusa meghatározható, azt tekinti a változó típusának.



45. ábra – ExportType

Az „exporttípusok” egy *ExportTypeDictionary* típusú objektumba kerülnek, mely asszociatív tárakkal tárolja el függvényszinten a változókat és azok *feltételezett* típusát. Maguk a típusok pedig egy *ExportType*-példányban vannak reprezentálva (45. ábra).

Az *ExportTypes* hat (+ egy) típusfeltételezést támogat: *Floating*, *Integer*, *Boolean*, *String*, *Complex*, *Array* (és *Unknown*), melyeket a *BasicType* enumeráció kódolja.

A *_recordName* mező mindig *null*, kivéve, ha a *Type* mező értéke *Complex*. Mivel a „complex” a rekord-típusokra utal.

Ha az *ExportTypeDictionary* felépítése kész van, példányosodik az absztrakt *Exporter* osztály valamelyik valós példánya. Ez az osztály foglalja össze mindazt, amivel egy *ExportTypeDictionary*ből és egy gyűjteményből elő lehet állítani egy kimeneti programozási nyelv kódját. Jelenleg két osztály öröklődik belőle: a *CSharpExporter* és a *PythonExporter*. Ezek rendre a C# és Python nyelv exporterei.

Jobbra, a 46. ábrán láthatók az osztály tagjai. A tagfüggvények sokasága miatt mindegyikre nem érdemes kitérni.

Legfelül a három privát adattag sorolja fel azokat az entitásokat, amelyek exportálásra kerülnek. A *_varTypes* mező az ezekhez az entitásokhoz tartozó típusinformációkat hordozza.

A legtöbb osztálytag jobb oldalt absztrakt; ezek mind az *Exporter* konkrét megvalósításaiban kerülnek implementálásra. Ezek tárgyalása a következő alfejezetben történik. Most viszont a *Print()* a lényeg. A függvény első paramétere az exportfájlt határozza meg. A megadott fájlba kerül – feldolgozás közben folyamatosan – a kimenet. Ha a fájl már létezik, felülírásra kerül.

Az exportálandó entitások kiválogatásra is itt kerül sor. A kiválogatást a *Traversals* osztály függvényei végzik, mely függvények különböző módon *(de mind fentről-lefelé)* járják be a szintaxisfát, hogy információt szolgáltassanak az exporternek.

Exporter

- _constantsToExport : Dictionary<string, object>
- _recordsToExport : List<RecordDefinition>
- _functionsToExport : List<BlockFunction>
- _varTypes : ExportTypeDictionary

+ Print(path : string, varTypes : ExportTypeDictionary)

Print(stream : StreamWriter, depth : int, ...)
PrintMainCode(stream : StreamWriter, depth : int)
PrintIf(...)
PrintWhile(...)
PrintReturn(...)
PrintBranches(...)
PrintAssignment(...)
PrintFunctionCallInst(...)

PrintExpression(exp : Expression, ...) : string
PrintSlice(...) : string
PrintBinOp(...) : string
PrintIndexer(...) : string
PrintLambda(...) : string
PrintVariable(...) : string
PrintUnaryOp(...) : string
PrintArrayExpr(...) : string
PrintConditional(...) : string
PrintExplicitCast(...) : string
PrintConstValue(...) : string
PrintFunctionCall(...) : string
PrintFormatString(...) : string
PrintListExpression(...) : string

PrintFunction(...) : string
PrintRecordDef(...) : string
PrintConstDef(...) : string

TransformExpression(...) : Expression
Pad(depth : int) : string

- Optimize(exp : Expression) : Expression

46. ábra – az Exporter felépítése

A *PrintExpression()* függvény szintén nem absztrakt. Ez felelős az *Expression* típusú objektumok stringgé való szerializálásáért. Emellett a szintén nem-absztrakt *Optimize()* függvényt is innen lehet elérni egy *bool*-paraméter által. Továbbá egy később részletezendő, fontos funkcióért is felelős, a *TransformExpression()* hívásáért is.

Az *Optimize()* függvény jelenlegi állapotában csak konstans kifejezéseket képes egyszerűsíteni. Pl. a `2 + 3` kifejezést `5`-re cseréli le.

Talán szembetűnő, hogy az osztálytagok között szerepel a *PrintSlice()*, *PrintLambda()*, *PrintConditional()*, *PrintExplicitCast()*, *PrintFormatString()*, és *PrintListExpression()*. Ezek rendre a *Slice*, *Lambda*, *Conditional*, *ExplicitCast*, *FormatString* és *ListExpression* típusú *Expression*ök szerializálásáért felelősek. Ezek az *Szintaxisfa és kifejezések* fejezet végén nem lettek feltűntetve, ugyanis a *BlockScript* nyelvnek nem részei, azonban az exportáláshoz szükségesek, ugyanis számos kifejezés csak ezek segítségével fordítható le. Csak az Exporterek által használatosak.

Ha már megvannak az exportálandó entitások, és a szükséges típusinformációk is, a *Print()* metódus meghívja a *PrintMainCode()* metódust, melynek paraméterként átadja a kimeneti adatáramot. A meghívott függvény pedig – implementációtól függően – elkezdi konvertálni (és közben kiírni) a kódot.

A *PrintFunction()*, *PrintRecordDef()* és *PrintConstDef()* eljárások a nekik megadott adatáramra írnak egy konkrét entitást.

A *PrintIf()* metódustól kezdve a *PrintFunctionCallInstr()* metódusig mind egy-egy *BlockScriptNode* kiírásáért felelősek.

A *PrintSlice()* és *PrintListExpression()* közötti függvények a különböző *Expression*ök stringgé való konvertálásáért felelősek, továbbá itt történik meg a kifejezések szintaxisfájának transzformálása a *TransformExpression()* függvény által. Erre azért van szükség, mert nem minden *BlockScript*-beli kifejezést lehet „egy az egyben” egy másik nyelv kifejezésére leképezni. Példa erre az `1 in { 1, 2 }` kifejezés, mely C# nyelvre így fordítható le:

new List<dynamic>() { 1, 2 }.Contains(1)

A két kifejezés szintaxisfája merőben eltér.

Az egyes exportereken belül definiálva van az adott nyelv precedencialistája, melyre azért van szükség, mert a *BlockCode* mindegyik nyelvtől némiképp eltérő precedenciákat használ.

A *Pad()* függvény szerkezeti segédfüggvény: a paraméterének megfelelő számú szóközből álló stringet ad vissza.

Beállítások

A szoftverben számos felhasználói beállítás van jelen. Ezek perzisztens tárolására a BlockCode a beállításokat a *Settings.json* fájlba menti közvetlenül a szoftver mellé. Mint arra a fájl nevéből következtetni is lehet, a konfigurációs fájl JSON formátumú. Benne az alábbi mezők találhatók:

Mező neve	Típusa	Leírása	Alapértéke
Theme	egész szám	A beállított téma azonosítója. (0, 1, 2 vagy 3)	0
HQRendering	logikai	Be van-e kapcsolva a jó minőségű renderelés.	igaz
ProgramDelay	egész szám	Az utasításkésleltetés ezredmásodpercekben.	100 ms
KeywordSet	string	A kulcsszókészlet neve.	”default”
LoadStandard	logikai	Betöltődjön-e az alapértelmezett könyvtár a *BlockCode* elindításakor.	igaz
IndexFromZero	logikai	Nullától történik-e a tömbök és stringek indexelése.	igaz
CustomKeywords	stringek tömbje	Az egyénileg beállított kulcsszókészlet. A kulcsszavak az alábbi sorrendben következnek: *If*, *Then*, *Else*, *ElseIf*, *EndIf*, *While*, *Do*, *Next*, *Function*, *EndFunc*, *Record*, *Const*, *And*, *Or*, *Xor*, *Impl*, *Not*, *New*, *Return*, *True*, *False*.	nincs

Az egyes mezők lehetséges értékei már le lettek írva. A kulcsszavak a *Language.KeysetWords* enumerációban vannak felsorolva.

A konfigurációs fájl a szoftverrel együtt töltődik be a *MainForm* konstruktorában. Ha nem létezik, automatikusan létrejön. Ha hibás, akkor a hiba jellegétől függően vagy alaphelyzetbe állnak a hibásan beállítások, vagy a hibás fájl törlődik, és új fájl jön létre. Bizonyos esetben programhibát okoznak. A fájl manuális szerkesztése nem ajánlott.

Tesztelés

Egy felhasználói szoftvert sem lehet tesztelés nélkül kiadni. A *BlockCode* tesztelésére megírt eljárások a mellékelt *Tester* projektben találhatók egységtesztek formájában. A tesztelést az *MSTest* könyvtár teszi lehetővé. A tesztosztályok a *TestClass* attribútummal, a tesztmetódusok a *TestMethod* attribútummal vannak ellátva. Ez biztosítja, hogy a teszteket a *Visual Studio* IDE-ben könnyen le lehessen futtatni. A „konkrét igazságvizsgálat” az *Assert* osztály metódusai segítségével történik.

A teszteket a Visual Studióban a **Tests** → **Run** → **All Tests** menüponttal lehet lefuttatni, vagy a *Test Explorer* segítségével.

Nyelvi tesztek

A szoftver leginkább tesztelendő egysége a *Language* projekt (lexer, parser), így ezen célra számos egységteszt került megírásra.

Az *ExpressionTests* tesztosztályban található három teszteset, melyek a kifejezések értelmezésére használt Shunting Yard algoritmust, és a zárójelezésre használt algoritmust tesztelik:

- *ParenthesesMismatch* – leteszteli, hogy hibás zárójelezésre ad-e hibát.
- *Parentheses* – azt teszteli le, hogy a fölösleges zárójelekkel is meg tud-e birkózni.
- *Parenthesizing* – a már meglévő szintaxisfák kiírásakor használt zárójelezést teszteli le.

A *ParsingTests* osztály a parsert teszteli le. Eldönti, hogy a megadott bemenetekre helyes eredményt ad-e a parser, vagy éppen azt, hogy hibát ad-e:

- *Elementary* – utasítások parse-olását teszteli le, mint pl. értékadás, visszatérés stb.
- *InnerBlocks* – vezérlési szerkezeteket tesztel, tehát elágazásokat, ciklusokat.
- *OuterBlocks* – a programfájlban közvetlenül előforduló szerkezeteket teszteli, tehát rekordok, konstansok és függvények/metódusok definiálását.

A *TokenizingTests* osztály egyetlen metódusa, a *TokenIndexing*. a lexer azon szempontból teszteli le, hogy az egyes tokenekhez megfelelő sor- és oszlopszámot társít-e. Ez lexikális, szintaktikus és szemantikus hibák esetén segíti a kódban való tájékozódást.

Futtatási tesztek

A szoftver egyik fő komponense a virtuális gép (*VirtualMachine*), mely a metódusok (*BlockFunction*) futtatásáért felelős. Ennek letesztelésére áll rendelkezésre a *RunningTests* osztály metódusa, mely a *Tester/TestCases* mappában található kódfájlokat tölti be egyenként, és teszteli le őket egy *VirtualMachine*-példányon a bennük található *Main* nevű metódus lefuttatásával. Mivel a *GlobalRecords*, a *GlobalFunctions* és a *GlobalConstants* osztályok statikusak, ezért minden teszteset lefuttatása után ki kell üríteni őket. A mellékelt tesztesetek a lehető legtöbb esetet lefedik.

A `read()` függvény természete miatt – mivel az felhasználói bemenetet vár – a *VirtualMachine.DialogParent* tulajdonságot (típusa *IDialogInput*) be kellett vezetni, hogy *.NET Core* alatt is lehessen *Form* nélkül bemenetet kérni. A teszt a *DummyDialogInput* implementációt használja.

A *TestCases* mappában minden kódfájl egy-egy szempontból teszteli le a futtatást. Az egyes fájlok és azok célja az alábbi listában olvasható:

- *Basic types.stpr* – a négy alaptípus (szám, logikai érték, string, tömb) működését vizsgálja különféle kifejezések kiértékelése során.
- *Built-in functions.stpr* – a beépített függvényeket teszteli le aszerint, hogy megfelelő eredményt adnak-e, és megfelelően reagálnak-e hibás bemenet esetén. Továbbá a „nem megfelelő argumentumszám” esetét is vizsgálja.
- *Corecion.stpr* – az operátorok ad-hoc polimorf viselkedését vizsgálja azáltal, hogy „ugyanazt” az operátort más-más típusú operandusokkal hívja meg.
- *If.stpr* – az egy- és többágú elágazásokat teszteli. Ez sokkal inkább a *Language* projektet teszteli, mint a *BlockCode*-ot.
- *Records.stpr* – a rekordok működését vizsgálja, beleértve az egyenlőségvizsgálatot.
- *Print method.stpr* – a `print()` metódus működését teszteli.
- *Copy and pass.stpr* – az értékátadásokat (érték szerinti és referencia szerinti) teszteli le.
- *Operators.stpr* – leellenőrzi az összes operátor helyes működését.

Megjegyzés: a tesztfájlok között – és a tesztfájlokon belül is – előfordulhat redundancia.

Egyéb tesztek, megjegyzések

Exportálás

A programnyelvekre való exportálás tesztelését az *ExportingTests* osztály végzi:

- *CSharpExporter* és *PythonExporter* – az előbb említett *Tester/TestCases* mappában található kódfájlokat tölti be, és konvertálja C# és Python nyelvre. Továbbá a beépített függvények és kifejezések exportálhatóságát is teszteli. Az elkészült fájlok átmenetileg elmentődnek.
- *TypeDeduction* – a függvények/metódusok lokális változóinak és formális paramétereinek típuskikövetkeztetésére szolgáló funkciókat teszteli le, és egyben az *ExportTypeDictionary* osztályt is teszteli.

Az egységtesztek lefutása előtt az *Initialize()* metódus fut le; a lefutás után a *CleanUp()* metódus törli az ideiglenes fájlokat, amennyiben a *DELETE_AFTER* konstans definiálva van.

Grafikus felület

A grafikus felülethez nem készültek automatikus tesztesetek; az kézzel került tesztelésre. A felület tesztelésekor a legfontosabb szempontok az alábbiak voltak:

- Eseménykezelők tesztelése. Az egyes *Control*ok a megfelelő callback-eljárást hívják-e meg, és ha igen, megfelelő módon módosítják-e a GUI többi elemét.
- A modellek állapotával konzisztens-e a felület, azaz a felülettel lehetetlen-e szabálytalan állapotba hozni a modellt. Rossz állapot az például, ha egy ciklus feltétel nélkül marad (*null*), vagy egy rekordnak mezőeltávolítás után nem marad több mezője.
- Ha a felület módosítása másik szálról történik, történik-e *Invoke*.
- Ha a program dolgozik, használható-e a felület, vagy éppen deaktiválva van-e az.
- Reszponzív-e a GUI. Más felbontások, más nagyítás, más színösszeállítás esetén is használható-e (és esztétikus-e).
- A szövegbuborékok (*ToolTip* vezérlők) megjelennek-e.

Debug-mód

A *Canvas* tesztelésére két debug-funkció került a szoftverbe. A *Canvas* bal felső sarkában az éppen betöltött struktogram szintaxisfája van lépcsőzetesen kiírva. Minden blokk bal felső sarkában megjelenik egy szám, amely az adott blokk (kvázi egyedi) hasítókódja (mely egyszerűen a *GetHashCode()* függvény visszatérési értéke).

A **Fájl → Debug** menüpontra kattintva (**Ctrl + Shift + D**) egy felugró üzenetben kilistázódnak a definiált függvények, rekordok és konstansok.

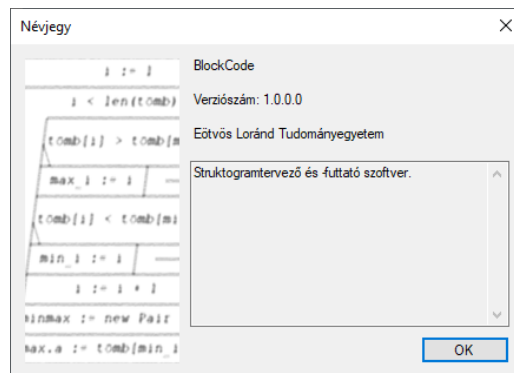
A leírt funkciók csak akkor érhetők el, ha a kód debug-módban – azaz a *DEBUG* konstans jelenlétével – fordul le. Továbbá a *Debug.WriteLine()*, *Debug.WriteLineIf()* és *Debug.Assert()* utasítások is csak debug-módban kerülnek bele a futtatható állományba.

Ezen kívül a forráskódban számos helyen szerepel még feltételesen lefordítandó kód, mely szintén csak a *DEBUG* konstans jelenléte esetén fordul csak bele a kódba. Például az *ExpressionParser* osztály feltételesen használja a *DebugStack* nevű speciális vermet, mely a sima veremre *(Stack)* épül, és minden *Push()* és *Pop()* műveletre (és a konstruktorra) debug-üzenetet ír ki.

Összefoglaló

Az előbbiekben a *BlockCode* felhasználói szoftverről, annak használatáról és működéséről olvashatott.

A felhasználói dokumentáció a felhasználói felülettől kezdve, a szoftver funkcióin át, a *BlockScript* nyelv leírásáig lefedte mindazt, amit egy felhasználónak tudnia érdemes a szoftver használatához.



47. ábra – a szoftver névjegye (F1 billentyű)

A fejlesztői dokumentáció a szoftver felülete mögé adott bepillantást, leírva annak implementációs részleteit. A GUI működésének leírásától kezdte, majd a végrehajtás részletezésén és a nyelvi reprezentációkon keresztül eljutott a tesztelésig. Kitért a felhasznált osztályok, interfészek és egyéb programozási eszközök felhasználási módjaira, hogy az olvasó pontos képet tudjon alkotni arról, mi is van a „motorháztető alatt”.

Személyes megjegyzések

A *BlockCode* megírásakor az elsődleges szempontok az átláthatóság, karbantarthatóság és „emberközeliség” volt; a sebesség csupán másodlagos, hisz a szoftver céljait, felhasználási eseteit végiggondolva nincs szükség optimalizálásra. A szoftvert több különböző korú gépen is leteszteltem, és a sebességgel nem volt probléma. A memóriahasználat elhanyagolható.

A szoftver noha kész állapotban van, rengeteg módon lehet még a funkcionalitását fejleszteni. A teljesség igénye nélkül hadd soroljak fel pár lehetőséget!

- FOR és FOREACH *(range-based for)* ciklus és hátultesztelős ciklus implementálása.
- A mentési fájlban explicit típusdeklarációk annotációk formájában, melyek nem befolyásolnák magukat a struktogramokat, azonban exportáláskor hasznosak lehetnének, hogy ne legyen szükség felhasználói bemenetre.
- Az exportálás előtti típusfelismerés segítése heurisztikákkal. Esetlegesen probabilisztikus típusmeghatározás, melyet a felhasználó kedvére (de)aktiválhatna.
- Több nyelvre való exportálás lehetősége.
- Új témák megtervezése és létrehozása.
- Verem és sor implementálása.
- A struktogramok szélességének manuális állítása.
- Objektumorientált programozásra való lehetőség kialakítása.
- A függvényhívási verem megtekintése és interaktív frissítése.
- A struktogramok állapotterének változásit nyomon lehetne követni táblázatos formában.
- A renderelés optimalizálása. A fabejárások számának csökkentésével.
- A kód „objektumorientáltságának” növelése, és ezzel a típusbiztonság növelése.

Továbbá szeretnék köszönetet mondani a szakdolgozatom megírásához való segítségért témavezetőmnek, Veszprémi Anna tanárnőnek, illetve minden további tanáromnak, akik munkája és segítsége miatt tudtam ezen szoftvert elkészíteni.