

A ScriptKernel programozási nyelv

Készítette: Baráth Áron
Programtervező Informatikus, MSc.
ELTE

Témavezető: Tóth Melinda
egyetemi tanársegéd
ELTE, Informatikai Kar,
Programozási nyelvek és Fordítóprogramok Tanszék



2013. január 07.

Absztrakt

Egy programozási nyelv szabályai befolyásolják annak használhatóságát. Ez főleg kezdő programozók körében nyilvánul meg, mert ha a nyelv túl elfogadó, akkor alapvetően hibás forrást is lefordít, ami futásidejű hibákhoz vagy hibás működéshez vezet. A megoldás egy olyan programozási nyelv, amely szigorú szintaktikai és szemantikai megszorításokat tesz, például a szigorú típusrendszer. Ilyen megszorítások mellett a fordítóprogram kevesebb forrásból készít futtatható programot, de ezek a programok megbízhatóbbak. A dolgozatomban bemutatom a saját fejlesztésű nyelvemet, amely szigorú szintaktikai és szemantikai szabályokkal igyekszik növelni a program megbízhatóságát.

Tartalomjegyzék

Bevezető	5
1. Nyelvi elemek	10
1.1. Vezérlő parancsok	11
1.2. Blokkok	14
1.3. Deklarációk	20
1.4. Operátorok	24
1.5. Vezérlési szerkezetek	26
1.6. Kifejezések	35
1.7. Értékadás	36
1.8. Kivételkezelés	38
2. Típusrendszer	43
2.1. Típusok	43
2.2. Típusok levezetése	50
3. Virtuális gép	54
3.1. A virtuális gép felépítése	54
3.2. Az assembly nyelv	56
3.3. A bináris állomány	57
3.4. Futtatás	60
4. Konstrukciós szempontok	64
4.1. Ötletek és megvalósítások	64
4.2. Algoritmusok	71
4.3. Eredmények	75

5. Összegzés	79
5.1. Elérhetőség	80
5.2. Továbbfejlesztési lehetőségek	81
Függelék A.	85
A.1. A nyelv BNF leírása	86
A.2. Operátorok	88
A.3. Névvel ellátott blokk használata	94
A.4. Elágazás példa elif nélkül	96
A.5. A foreach használata	97
A.6. Példa operátor túlterhelésekre	99
A.7. Lokális változók létrehozása	102
A.8. Szabványos kivételek	104
A.9. Lusta „és” és lusta „vagy”	107
A.10. Gyorsrendezés	110
A.11. Külső C-s könyvtárak felhasználása	114
A.12. A virtuális gép utasításkészlete	117
A.13. Példa algoritmus implementálásra	123
A.14. Képernyőkép a pályaszerkesztőből	124

Bevezető

Dolgozatom témája egy saját fejlesztésű erősen típusos programozási nyelv, mely jelenleg a 0.7.9.9-es verziónál jár. A nyelv szintaxisa a C nyelv szintaxisára épül, de más programozási nyelvekből is merít ötleteket. A módszertanból megismert eszközök nagy részét támogatja – például a tömb adatszerkezet helyett csak sorozatokat lehet létrehozni. A nyelv legerősebb eszköze a statikus típusrendszer, és a jelenlegi változatban dinamikus típusok nem is léteznek. További szigorításokkal próbálja a szintaxis is segíteni a programozót, hogy minél kevesebb hibát követhessen el a forráskód írása közben. A programok egy virtuális gépen futtathatók, amely szintén saját fejlesztésű, továbbá készült hozzá egy assembly nyelv is. A programok összeszerkesztése dinamikusan történik a típusok (függvényszignatúrák) egyeztetésével – ezáltal az összeszerkesztés után sem sérülhet a statikus típusozás.

A nyelv erősségei közé tartozik még a könnyű átláthatóság, amit a szintaxis és a szematikai szabályok kényszerítenek ki. A sok megszorítás ellenére használható a függvény- és operátor túlterhelés mechanizmusa is, ami a tömörebb programok írásához elengedhetetlen. Céлом az, hogy minél szélesebb körben elterjedjen, és bekerüljön az egyetemi kurzusok közé is, mert kezdő és haladó programozók szempontjából is előnyös tulajdonságokkal rendelkezik.

Motiváció.

A gimnáziumi évek alatt a szabadidőmben egy autós játék fejlesztésével foglalkoztam. Ennek kapcsán merül fel az igény arra, hogy lehessen szkriptelni a játékot. A szkriptelés alatt az autók paraméterezését,

a felhasználói felület elkészítését és megjelenítését, események kezelését (például: mi történjen, ha a játékos megnyeri a versenyt) értem. Úgy döntöttem, hogy nem meglévő nyelvet használok fel, hanem készítetek egy újat, ami megfelel a saját igényeimnek – pontosan annyit tudjon, amennyit kell, nem kevesebbet, nem többet, valamint olyan szintaxist készítetek, amilyen tetszik. Ennek az az oka, hogy a meglévő szkriptnyelvek közül egyik se nyerte el a tetszésemet.

A Lua szintaxisa és működése se nem szimpatikus, se nem előnyös (szemétygyűjtés van benne, ami gyakori nagymértékű megtorpanást okoz – a teljesítményteszt alapján). A JavaScript szintaxisa már rendben van, de túl nagy az értelmezője, és nem játék szkriptelésre való. Az UnrealScript egy nagyon jó szkriptnyelv, de sajnos csak az Unreal Engine-ben használható. A REXX nyelv lehetőségei túlságosan korlátozottak, ezért ez a nyelv számításba se jöhetett, valamint nem játékszkriptelésre való.

Az eredeti, kezdetleges változatnak még a szintaxisa sem hasonlított a jelenlegihez, az elágazás, valamint az egy féle ciklus is limitált darabszámú utasítást tartalmazhatott, valamint a típusrendszer sem volt annyira szigorú. A gyakorlat megmutatta, hogy ez a koncepció nem életképes, mert amikor odaadtam a csapattársaknak a nyelvet, akkor problémákba ütköztek, és rengeteg futásidejű hiba történt (ez rendszerint típushibát jelentett).

Az új szintaxis, és az új típusrendszer úgy lett kifejlesztve, hogy a nyelv a lehető legszigorúbban ellenőrizze a leírtakat, és minél több programozás hibát derítsen fel a fordító. Így, a fordító által elfogadott programoknak sokkal nagyobb része működött kifogástalanul, mint az első változatban. Az első változat forrásai a **#version 1.0** szignatúrával kezdődtek, és ezért kell **#version 2.0** szignatúrával kezdődnie az új verzió forrásainak (*ld. 1.1.1. fejezet*).

A nem programozó csapattársak körében siker koronázta a szigorú szintaktikai és szematikai szabályokat alkalmazó nyelvet. Ezért elkezdtem bővíteni olyan elemekkel, melyeknek forrása a módszertan vagy más programozási nyelv. Egy új funkciót csak akkor építettem bele a

nyelvbe, ha az a szigorú statikus típusrendszert nem tudta kijátszhatóvá tenni.

A folyamatos fejlődés miatt a nyelv kinőtte magát egy játék script nyelvéből egy sokkal nagyobb, és sokkal komolyabb funkciók szolgáltató nyelvvé. Az elnevezés is az eredetéhez kapcsolódik: az alapötlet az volt, hogy egy olyan nyelvet kell készíteni, amit könnyen ki lehet bővíteni az aktuális játékra jellemző C-s függvényekkel, és így lehet scriptelni egy-egy játékot. Emiatt lett a neve *ScriptKernel*: egyrészt script nyelv volt (*Script*), másrészt pedig csak egy séma volt, amit könnyen ki lehet egészíteni (*Kernel*). A nyelv gyorsan változott, és mire ráeszméltem arra, hogy a jelenlegi név már nem helytálló, addigra már számos helyen szerepelt a név, és ezeknek a lecserélése már túl költséges lett volna. Ezért megmaradt a *ScriptKernel*.

A nyelv sok módszertanból megismert (az Eötvös Loránd Tudományegyetemen Fóthi Ákos által tanított programozási módszertan [3]) funkciót tartalmaz változtatás nélkül (ugyanaz a szintaxis és a szemantika), ezért célom az, hogy a nyelvet az egyetemi képzés részévé tegyem. Már a jelenlegi változat is képes lenne kitölteni egy BSc-s tárgyat. Rengeteg segítséget jelentene a programozás elsajátításában, hiszen a szintaxis módosításával nem kell foglalkozni. Egy absztrakt megoldás kódolása általában annyit jelent, hogy más szintaxissal kell leírni ugyanazt. A feladat önmagában nem nehéz, de ha mellette még a megoldást is el kell készíteni, az már sok esetben problémát tud okozni az egyetemet kezdő hallgatók körében. Az egyetemen a C++ nyelvet oktatják az első hallgatónak is, ami magában rejt néhány nehézséget: a sorozatokat (tömböket) más módon kell használni a struktogramban, és máshogyan C++ nyelven.

A nyelv fejlesztése jelenleg az általános célú felhasználás felé mozdult el. Fő cél a lehető legbiztonságosabb szintaxis és szemantika annak érdekében, hogy a fordító által elfogadott források legnagyobb része helyes legyen.

Dolgozat eredményei.

A dolgozatom témája egy saját, biztonságos magasszintű programozási nyelv, amely illeszkedik az ELTE-n tanított programozási módszertanhoz. A szigorúan statikus típusrendszernek több előnyét is bemutatom, köztük a dinamikus, típus-biztos linkelés lehetőségét. A nyelv támogatja a függvényszintű kivételkezelést is.

A magasszintű nyelv kimenete egy saját assembly nyelv forrás, amit az assembler programom feldolgozva előállítja a futásra kész bináris állományt. Az assembly nyelv a tartalmaz típus információkat, amelyek a betöltés során szükségesek. A betöltés során minden exportált függvényt és felhasználói típust ellenőriz a betöltő: függvény esetén a névnek, az argumentumok számának és típusának, és a visszatérési értékek számának és típusának egyezése; rekord esetén a névnek, a mezők darabszámának és típusának egyezése szükséges a sikeres betöltéshez.

Ez az assembly a saját fejlesztésű virtuális gép vezérlőnyelve. A virtuális gép betöltője a létrehozott bináris állományt képes betölteni. A virtuális gép része a végrehajtási környezet, amelybe a programok képesek függvényeket, típusokat és kivételeket exportálni, és a betöltés során ebből a környezetből szolgálja ki a programok importálási igényeit a betöltő. A virtuális gép nem csak a program belépési pontját (**main**) képes végrehajtani, de bármely, a környezetből elérhető függvényt is lehet futtatni¹.

A nyelv alkalmazható az oktatásban is, hiszen a szintaxisa illeszkedik az ELTE-n tanított módszertanhoz. Az alkalmazási terület nem korlátozódik az oktatásra, mert tetszőleges C vagy C++ függvényeket be lehet illeszteni a virtuális gép környezetébe, és ezáltal a ScriptKernelhívhat C/C++ függvényeket, de a C és C++ függvények is hívhatnak ScriptKernelfüggvényeket a virtuális gépen keresztül.

Dolgozat összefoglalása.

A dolgozatban bemutatom a nyelv szintaxisát a nagy egységektől egészen a részletekig. A szintaxis kapcsán bemutatásra kerül a vezér-

¹Abban az esetben, ha a függvény paramétereit be lehet állítani az API segítségével.

lési szerkezetek és a kivételkezelés is. Minden nyelvi elemhez tartozik egy rövid szemantikai leírás is (*ld. 1. fejezet*).

A nyelvi elemekről szóló fejezet után a részletes, szemantikával foglalkozó fejezet következik, amelyben a típusokat és a fordítóprogramban használt típusellenőrzést mutatom be (*ld. 2. fejezet*).

A következő fejezet a fordítóprogram kimenetének két formáját mutatja be (assembly forrás és futtatható állomány), valamint a virtuális gép felépítését és alapvető mechanizmusait. A fejezet végén található a lefordított programok futtatásának a leírása (*ld. 3. fejezet*).

A negyedik fejezet a nyelvben használt konstrukciók háttéréről, használhatóságával és értékeléséről szól. A saját nyelvet összehasonlítom más nyelvekkel (*ld. 4. fejezet*).

Az ötödik fejezet tartalma a dolgozat összefoglalása és a továbbfejlesztési lehetőségek leírása – a nyelv következő verziójában a felsorolt változtatások egy része szerepelni fog (*ld. 5. fejezet*).

Az utolsó fejezet a függelék, ahol több komplett példaprogram is található, valamint a virtuális gép utasításkészlete és az utasítások rövid leírása (*ld. A. fejezet*).

1. fejezet

Nyelvi elemek

Ebben a fejezetben a magasszintű nyelv szintaxisáról és annak szemantikájáról lesz szó. A szintaxis alapvetően C-szerű, de megtalálhatók benne például Adás vonások is. Ez a nyelv egy imperatív procedurális nyelv.

A nyelv legmagasabb szintaktikai egysége a blokk és a forráskód/program vezérlő parancsok. Minden további szintaktikai egység a blokkokon belül helyezkedik el.

Egy utasítás lehet bármely vezérlési szerkezet, értékadás¹, függvényhívás, vagy blokk utasítás.

A programszöveg blokkokból és vezérlő parancsokból (*ld. 1.1. fejezet*) áll. A vezérlő parancsokkal metaadatokat lehet hozzáadni, valamint néhány beállítást lehet elvégezni. A blokkok utasítások, változó deklarációkat, függvény szignatúrákat, kivétel azonosítókat és típus leírásokat tartalmaznak. A legfontosabb blokk a **main** blokk (*ld. 1.2.7. fejezet*) – a programszövegnek tartalmaznia kell pontosan egy **main** blokkot. A függvény definíció egy speciális blokk, a **function** blokk (*ld. 1.2.8. fejezet*). A blokk fejléce a függvény szignatúrája, amivel a fordító hozzárendeli egy névhez (és argumentum típusokhoz) a blokkban található utasításokat. Az utasításokat tartalmazó blokkok többségében el lehet helyezni kivételkezelő részeket is (ez alól kivétel a **ctor** és a **dtor** blokk

¹Az értékadás nem kifejezés, mint a C++ nyelvben, hanem utasítás, mint az Ada nyelvben.

– a „*constructor*” és a „*destructor*” után). Az utasítások között szerepelhetnek lokális változó deklarációk, értékadások és különféle vezérlő utasítások (elágazás, ciklusok). A BNF leírás megtalálható a függelékben (ld. A.1. fejezet).

1.1. Vezérlő parancsok

1.1.1. Aláírás

Minden forráskódnak az alábbi aláírással kell kezdődnie:

```
#version 2.0
```

Az aláírás előtt semmi sem szereplhet!

Ezt a fordító ellenőrzi, és ha nem ezt találja ott, akkor a fordítás megáll. Ez a megszorítás az „*include*”-olt forrásokra nem vonatkozik – sőt, ott hibának tekinti a fordító az aláírást.

1.1.2. Forráskezelés

Két parancs használható a források kezeléséhez. Az elsővel más forrásokat lehet bemásolni a parancs helyére:

```
#include "filename.inc"
```

A forrás beillesztése során semmilyen további ellenőrzés nem történik (pl. nem ellenőrzi, hogy a nyitó illetve csukó kapcsolósárójelek száma megegyezik-e). A hatása csupán az, hogy a lexikális elemző egy másik forrásban folytatja a lexikális egységek felismerését, és ha az véget ért, akkor folytatja ott, ahol a **#include** volt.

Ez a mechanizmus hasonlít a C preprocesszor **#include** direktívájához, a különbség csupán annyi, hogy a források beillesztése nem külön lépés, hanem a lexikális elemző átlép egy másik forrásba.

Az **#include** parancs nem ellenőrzi, hogy az adott forrás már be volt-e illesztve korábban. Ezt a feladatot a következő paranccsal lehet vezérelni:

```
#once SomeIdentifier
```

A fordító ellenőrzi, hogy a megadott azonosító szerepel-e a nyilvántartásában. Ha nem létezik ilyen azonosító, akkor eltárolja az új azonosítót (a fenti példában a **SomeIdentifier**-t), és folytatódik az elemzés. Ha létezik már a megadott azonosító, akkor a forrás **#once** utáni részét figyelmen kívül hagyja. Ezt a lépést nem jelzi sehol, mert nem hiba.

1.1.3. Verem méretek

A lefordított program végrehajtása két verem segítségével történik (*ld. 3. fejezet*). A program forráskódjában megadható az adatverem és a hívási verem mérete is. Ha nincs ilyen beállító parancs, akkor az alapértelmezett értéket fogja használni. Ez az érték az adatverem esetében 32768 byte, a hívási verem esetében pedig 8192 byte.

Az adatverem méretét átállítani a

```
#datastacksize 32768
```

vagy a

```
#dss 32768
```

paranccsal lehet.

Ugyanez a hívási verem esetében a

```
#callstacksize 8192
```

vagy a

```
#css 8192
```

parancs.

A többszöri értékbeállítás nem okoz hibát, az eltárolt, és később felhasznált érték az utolsó beállítás paramétere lesz. A méretekre nincs limit, a paraméter pedig négyféle egész szám lehet: előjeles decimális, előjeles hexadecimális, előjel nélküli decimális és előjel nélküli hexadecimális (*ld. 2.1. fejezet*).

1.1.4. Metaadatok

A program bináris állományában lehet tárolni metaadatokat is, melyek kulcsa is és értéke is szöveg. Van néhány előre definiált metaadat kulcs, de lehet egyedi kulcsokat is megadni.

A metaadatokban megadott értékeket a futtatókörnyezet nem ellenőrzi, és nem feltételez róluk semmit. Ezek az információk csak egyéni felhasználás céljából szerepelnek a programokban. A kulcsokhoz tartozó értékeket a futtató programmal ki lehet íratni, valamint a virtuális gép API-jában létezik lekérdező függvény.

A kulcs valamint az érték bármilyen szöveg (karakterlánc) lehet, amit a lexikális elemző felismer. Ide sorolható a több részből álló szövegek, valamint az elemzés során, a környezettől függően behelyettesített szövegek is (*ld. 2.1.1.1. fejezet*).

A következő metaadat a program készítőjének a nevét tárolja.

```
#meta:author "My Name"
```

Program neve:

```
#meta:title "My Program"
```

Program verziója:

```
#meta:version "1.1"
```

Másolásvédelmi jog információ:

```
#meta:copyright "GPL"
```

Verziókezelő rendszer Id kulcsszava.

```
#meta:id "$ any subversion Id keyword $"
```

Megjegyzés, tudnivalók:

```
#meta:comment "Comment string." " * Multi-part string."
```

Saját kulcs és érték:

```
#meta:* "Key" = "Value"
```

A fordítás időpontját el lehet tárolni metaadatként:

```
#meta:* "Build" = __DATETIME__
```

1.2. Blokkok

Minden legfelső szintű blokknak van egy speciális neve, ami meghatározza a blokk szerepét. A blokk törzse minden esetben { és } között helyezkedik el. A blokkok sorrendje fontos, mert az egyes blokkok (főleg függvény blokkok) hivatkozhatnak egymásra². A sorrend alacsonyabb szintű blokk esetén is számít (ld. 1.5.2. fejezet), mert a leírás sorrendjében fognak lefutni.

A legfelső szintű blokkokat a következő kategóriákba lehet sorolni:

- Deklarációkat tartalmazó blokkok

– declare

²A **declare** (ld. 1.2.1. fejezet) blokkot a forrás elejére kell írni ahhoz, hogy a megfelelő módon fejtse ki hatását.

- **import**
- **export**
- (Változó) definíciókat tartalmazó blokkok
 - **global**
- Utasításokat és változó definíciókat tartalmazó blokkok
 - **ctor**
 - **dtor**
 - **main**
 - **function**

Az első két kategóriába tartozó blokkok törzse nem lehet üres. Ennek oka az, hogy ha nem kell semmit sem importálni, akkor fölösleges leírni az üres blokkot. Azonban az utasításokat tartalmazó blokkok esetében van (lehet) értelme az üres törzsnek – például a még nem implementált függvények létezhetnek, és a program fordítható lesz.

Továbbá az első két kategóriába tartozó blokkoknak olyan tulajdonságuk is van, hogy a tartalmuk összevonásra kerül. Tehát több **declare** blokkot is el lehet helyezni a forrásban, és ezeknek a blokkoknak a tartalma összeadódik. Egy precízebb megfogalmazás: az azonos típusú blokkok tartalmát uniózza a fordító, és azt tárolja el. Ez azt jelenti, hogy akár egyazon **declare** blokkon belül többször is leírható egy függvény deklarációja úgy, hogy a fordító nem ad hibaüzenetet. Kivételt képez a **global** blokk, mert egy definíció csak egyszer szerepelhet – nem lehet értelmezni a többszörös definíciót.

1.2.1. A **declare** blokk

Ebbe a blokkba függvény, kivétel vagy rekord deklarációkat lehet elhelyezni, és legalább egyet. Az ide leírt deklarációk csak a programban használhatóak, a definíciójuk (függvények esetében) a programban kell lenniük (*ld. 1.3. fejezet*).

Ha kivétel deklaráció van ebben a blokkban, akkor a fordító azt exportálni is fogja, és egy figyelmeztető üzenet („*warning*”) formájában értesíti a felhasználót. Erre a kivételek egyértelmű azonosítása³ miatt van szükség (ld. 1.8. fejezet).

Ez a blokk rekurzív függvények esetében nélkülözhetetlen, mert enélkül a függvény deklarációja csak a definíció után kerül be a felhasználható függvények közé. Másik felhasználási terület a programban használható függvények felsorolása, hogy a definíciók sorrendje ne okozzon problémát.

1.2.2. Az **import** blokk

Hasonlóan a deklarációs blokkhoz, ide is deklarációkat lehet elhelyezni (legalább egyet), de ezeknek a deklarációknak más jelentése van. Ugyanúgy használhatóak a programon belül, mintha „sima” deklarációk lennének.

Importált függvény lehet bármilyen más program által exportált függvény, vagy a kontextushoz hozzáadott C függvény. A C függvények csak a beépített, egyszerű típusokat használhatják⁴.

Az importált kivételek lehetnek szabványos kivételek (pl.: ***IndexOutOfRangeException***, ***ConstraintErrorException***), más program által exportált kivételek, vagy a kontextushoz külön hozzáadott kivétel⁵.

Típusok (jelen esetben rekordok) importálása csak más program által exportált típusok közül lehetséges, mert semmilyen támogatás nincs a virtuális gép API-jában típusok deklarálására.

1.2.3. Az **export** blokk

Ennek a blokknak a tartalma legalább egy valamilyen deklaráció. Ezeket a deklarációkat a program a betöltés során exportálni fogja,

³A kivételnek egy globálisan egyedi azonosítót kell kapnia, amit automatikusan kap, ha exportált kivételről van szó.

⁴Ez magába foglalja az összes alaptípust, de a sorozatokat már nem tartalmazza.

⁵A virtuális gép API-ja támogatja a kivételek hozzáadását a kontextushoz.

azaz a kontextushoz hozzáadja a deklarációkat.

Az exportált deklarációk egyben normál deklarációk is, vagyis úgy használhatóak a programban, mint a közönséges deklarációk.

Külön megszorítás a függvények esetében, hogy a függvény törzsét el kell helyezni a programon belül, valamint a felhasznált nem alaptípusokat is exportálni kell (vagy importálni kell lennie).

1.2.4. A **global** blokk

A **global** blokkban változó definíciókat lehet elhelyezni, de legalább egyet. A többszörös definíció nem engedélyezett, még akkor sem, ha más a típusa. A változó deklarációnak meg kell felelnie az általános változó deklarációnak (*ld. 1.3.4. fejezet*).

A változóknak lehet kezdőértéket adni, ami lehet literál vagy egy függvény visszatérési értéke is. Ha nincs kezdőérték megadva, akkor a primitív típusok nullára inicializálódnak, az összetett típusok pedig üresre (*ld. 2.1. fejezet*).

Az itt felsorolt globális változókra a **global** minősítéssel is lehet hivatkozni, ha az aktuális scope-ból éppen nem látszik egy globális változó (*ld. 1.7.1. fejezet*), (*ld. 1.6. fejezet*).

1.2.5. A **ctor** blokk

Ez a blokk utasításokat és lokális változó deklarációkat tartalmazhat. Ennek a blokknak a tartalma közvetlenül a program betöltése után fut le, és csak egyszer. Ha a végrehajtás során kivétel lép fel, akkor a programot kitörli a virtuális gép, és úgy tekint a program betöltésre, mintha sikertelen lett volna.

Ebbe a blokkba olyan inicializálások elhelyezése ajánlott, ami a program futása során végig használtak, és a felszabadítások a program futása után kell, hogy lefussanak.

Ezt a blokkot nem kötelező megadni (opcionális). Valójában majdnem minden esetben lesz konstruktor a programban, amit lefuttat a

program betöltése után. Ezért volt ennek a blokknak a neve a korábbi verziókban **userctor**.

```
ctor
{
    // ...
}
```

1.2.6. A dtor blokk

Ez a blokk a **ctor** párja, ami szintén utasításokat és lokális változó deklarációkat tartalmazhat. Ennek a tartalma közvetlenül a program törlése előtt hajtódik végre.

Ezt a blokkot sem kötelező megadni. Ha a fordító készített konstruktor (ami nem a **ctor** volt), akkor lesz automatikusan generált destruktort is. A korábbi verziókban **userdtor** volt ennek a blokknak a neve.

```
dtor
{
    // ...
}
```

1.2.7. A main blokk

Az összes blokk közül ez az egyetlen, amelyet kötelező megadni – lehet üres is, de minden esetben meg kell adni. Ez a blokk utasításokat és lokális változó deklarációkat tartalmazhat. Ennek a tartalma a **ctor** és a **dtor** között fog lefutni.

Ez a blokk a program belépési pontja. Ez a blokk fog lefutni, ha a virtuális gép futtatja a programot⁶.

⁶Lehetőség van a program bármely, egyszerű típusokat használó függvényének a futtatására is a virtuális gép API-jában.

```
main
{
    // ...
}
```

1.2.7.1. Példa

Az alábbi program kiírja a szabványos kimenetre a „*Hello World!*” szöveget, valamint egy újsor karaktert.

```
#version 2.0
main
{
    write("Hello World!\n");
}
```

1.2.8. A function blokk

A nem importált függvény deklarációhoz tartozó törzs (definíció). Ez a blokk utasításokat és lokális változó deklarációkat tartalmaz. Ennek a tartalma egy függvény utasításai. A függvény meghívásakor ezek az utasítások fognak lefutni.

A függvény blokk címkéje az a függvény deklaráció, amelyhez a törzs tartozik (*ld. 1.3.1. fejezet*), de a deklarációt záró pontosvessző el van hagyva.

```
function name(<arglist>) <opt_retlist>
{
    // ...
}
```

1.3. Deklarációk

1.3.1. Függvény

A függvények deklarációjának tervezése során figyelembe kellett venni azt a tényt is, hogy egy függvénynek nem csak egyetlen érték lehet a visszatérési értéke, hanem akármennyi. A C stílusú függvény deklarációt nem találtam elég rugalmasnak ahhoz, hogy a több visszatérési értéket elegáns módon le lehessen írni, ezért egy másik nyelv ötletét használtam fel. Ez pedig az Adából ismert **function** kulcsszóval bevezetett deklaráció.

A deklaráció a **function** kulcsszóval kezdődik, utána a függvény neve, majd a paraméterlista (a nyitó- és csukó gömbölyű zárójel kötelező), végül a visszatérési értékek típusának felsorolása következik.

```
function name(<arglist>) <opt_retlist>;

<arglist> ::= <arg> | <arg> "," <arglist>
<arg>      ::= <type> <identifier>

<opt_retlist> ::= <<empty>> | ":" <retlist>
<retlist>    ::= <ret> | <ret> "," <retlist>
<ret>        ::= <type>
```

A típus bármilyen típus lehet, de exportált függvény esetében biztosítani kell a felhasznált típusok exportálását is⁷, valamint importálásnál a típusok importálását.

A nyelvben az operátorok speciális függvényként vannak megvalósítva, amit azt jelenti, hogy az operátor függvényt speciális módon is meg lehet hívni. A speciális mód a megszokott prefix (pl.: !, ~) vagy infix (pl.: +, *) alak. Az operátor függvény nevét idézőjelek között kell

⁷Amennyiben a típus nem alaptípus (ld. 2.1. fejezet).

megadni (pl.: "!", "+", "*")⁸. Ez a speciális jelölés használható „sima” függvények esetén is:

```
function func(i32 x, i32 y) : string, bool, u32;
```

helyett használható a következő is

```
function "func"(i32 x, i32 y) : string, bool, u32;
```

A két forma ekvivalens, és a két jelölés vegyesen is használható a függvény túlterhelés során. Az operátorok esetében csak az idézőjeles forma használható⁹.

A deklarációhoz tartozik definíció is, amennyiben nem importált a függvény (*ld. 1.2.8. fejezet*).

A függelékben található egy példa az operátor túlterhelésre (*ld. A.6. fejezet*).

1.3.2. Kivétel

A nyelvben a kivételek reprezentálása egyedi azonosítóval történik. Ezek az azonosítók exportálhatóak és importálhatóak egy kontextuson belüli programokból illetve programokba. Az export/import mechanizmus segítségével lehet elérni azt, hogy egy adott azonosító minden programon belül ugyanazt a kivételt jelentse. A kontextus a kivétel nevét is tárolja – szüksége van a névre is, mert így oldja meg a kivételek összekapcsolását. A virtuális gép API-jában van egy segédfüggvény, amivel le lehet kérdezni azt a kivétel azonosítót, amihez a megadott dinamikusan kiosztott kivétel azonosító tartozik.

A háttérben minden kivétel kap egy egyedi számot, amit dinamikusan rendel hozzá a virtuális gép a program betöltése során. Egy programot csak akkor lehet betölteni a kontextusba, ha minden importálandó

⁸Funkcionális nyelvekben (pl.: *Erlang*) is így, vagy hasonló módon kell megadni az infix operátorokat.

⁹Ez az operátor-jelölés az Adából származik, és a nyelvtan egyszerűsítése miatt van szükség az idézőjeles alakra.

hivatkozást fel tud oldani, és minden exportálást el tud végezni. Így a program csak akkor lesz futtatható, ha minden kivételhez megtalálta a dinamikusán kiosztott azonosítót.

Minden programnak van egy lokális „kivételfordító” táblázata, amit a lokális kivétel sorszámaival indexel¹⁰. A táblázat egyes elemei pedig a dinamikusán kiosztott kivétel azonosítókat tartalmazzák. Vagyis, amikor a programnak szüksége van egy globális¹¹ kivétel azonosítóra, akkor a kivétel táblázatából kiolvassa a lokális kivétel azonosító segítségével az értékét.

Kivételt deklarálni a következő módon lehet:

```
exception DummyException;
```

Ebben a példában importálásra vagy exportálásra fog kerülni a **DummyException** nevű kivétel. A kivétel deklarációkat a **import** vagy **export** blokkokba lehet elhelyezni.

1.3.3. Rekord

A rekorddal lehet felhasználó típusokat bevezetni a nyelvben. Egy rekordnak van neve, és vannak típussal ellátott mezői. Nem lehet diszkriminánsa (pl.: Ada), és nem lehet tagfüggvénye (pl.: C++). Minden rekordból lehet sorozatot készíteni (*ld. 2.1.2. fejezet*).

Rekordok értékadásakor olyan kód készül, ami a rekordot teljes egészében lemásolja¹². A rekordot paraméterátadáskor nem másolja, hanem referencia szerint adja át a függvénynek. Az alapértelmezett paraméterátadás a konstans referencia, és a paraméterül kapott rekordot a függvényen belül módosíthatóvá¹³ a **mutable** módosítóval lehet¹⁴, amit a típus neve elé kell írni. Tehát a deklaráció során semmilyen kiegészítő függvény elkészítésére sincs szükség. Ez a későbbiekben ki fog

¹⁰A lokális sorszámok minden programra rögzítettek, és nem lehet megváltoztatni.

¹¹Csak az adott kontextusban globális.

¹²*Deep-copy*.

¹³Csak módosítható, nem állhat értékadás baloldalán!

¹⁴Ez vonatkozik a sorozatokra és a szövegekre (**string**) is.

egészülni egy opcionális **function invariant(T val) : bool;** függvénnyel, ami a rekord invariánsát fogja ellenőrizni.

A rekord deklaráció felépítése:

```
record name
{
    <fields>
}

<fields> ::= <field> | <field> <fields>
<field>  ::= <type> <identifier> ";"
```

Ahol a <type> bármilyen típus lehet, és az <identifier> bármilyen azonosító, de a rekordon belül egyedinek kell lennie – ez az azonosító lesz a mező azonosítója (*ld. 2.1. fejezet*).

1.3.4. Változó

Változókat deklarálni csak a **global** blokkban, vagy utasítás blokkban lehet, a formális paraméterek deklarációját pedig a függvény deklarációja tartalmazza (*ld. 1.3.1. fejezet*).

A deklaráció általános felépítése:

```
<type> <identlist> <opt_assign> ";"

<identlist>  ::= <ident> | <ident> "," <identlist>
<opt_assign> ::= <<empty>> | "=" <expr>
```

A **<type>** bármilyen típus lehet (*ld. 2.1. fejezet*), az **<ident>** egy azonosító, és az **<expr>** egy kifejezés (*ld. 1.6. fejezet*).

Ha van kezdő értékadás, akkor a megadott kifejezés típusa meg kell egyezzen a deklaráció azonosító listájának a típusával. Ez minden esetben egy többértékű kifejezést jelent, amely típusa annyi darab **<type>** típus, amennyi eleme van az azonosító listának. Konkrét példa a függvényekben (*ld. A.7. fejezet*).

1.4. Operátorok

Az operátor is függvény, csak van egy speciális jelölési formája is. Ennek köszönhetően – a lusta operátorokat kivéve – minden operátort túl lehet terhelni¹⁵. Az operátor paraméterszámára nincs megkötés, de a kényelmes használathoz meg kell tartani a megszokott paraméterszámot¹⁶. A függelék egyik példája bemutatja, hogy milyen módon lehet a nem-szokásos paraméterszámú operátorral dolgozni (*ld. A.6. fejezet*).

Két lusta operátor létezik (mindegyikből csak egyetlen változat van), és ezeket nem lehet túlterhelni. Ennek az az oka, hogy csak logikai kifejezések esetén lehet konzisztens módon értelmezni a lusta kiértékelést (*ld. 1.4.1.1. fejezet*).

Minél kisebb egy operátor precedenciája, annál jobban „köt”, vagyis azt fogja később kiértékelni a végrehajtás során.

A bináris operátorok kiértékelése minden esetben balról jobbra történik, tehát először a baloldali argumentumot értékeli ki, utána pedig a jobboldali argumentumot. Ezután végzi el az operátor műveletét a kiszámított argumentumokon. Ez a lusta operátorokra is igaz azzal a különbséggel, hogy a jobboldali argumentumot nem biztos, hogy ki kell értékelnie.

1.4.1. Infix operátorok

Az összes előre definiált infix operátornak két ugyanolyan típusú paramétere van. A függelékben megtalálható az összes előre definiált operátor (*ld. A.2. fejezet*).

|| [Precedencia: 5, Asszociativitás: bal] Logikai vagy.

&& [Precedencia: 6, Asszociativitás: bal] Logikai és.

| [Precedencia: 7, Asszociativitás: bal] Bitenkénti vagy.

¹⁵A felüldefiniálás nem megengedett.

¹⁶Infix operátor esetén 2, prefix operátor esetén 1 paraméter szükséges.

$\wedge$ [Precedencia: 8, Asszociativitás: bal] Bitenkénti kizáró vagy.
 $\&$ [Precedencia: 9, Asszociativitás: bal] Bitenkénti és.
 $==$ [Precedencia: 10, Asszociativitás: bal] Egyenlőségvizsgálat.
 $!=$ [Precedencia: 10, Asszociativitás: bal] Nemegyenlőség-vizsgálat.
 $<$ [Precedencia: 11, Asszociativitás: bal] Kisebb.
 $>$ [Precedencia: 11, Asszociativitás: bal] Nagyobb.
 $<=$ [Precedencia: 11, Asszociativitás: bal] Kisebb vagy egyenlő.
 $>=$ [Precedencia: 11, Asszociativitás: bal] Nagyobb vagy egyenlő.
 $\ll$ [Precedencia: 12, Asszociativitás: bal] Balra „shift”-elés.
 $\gg$ [Precedencia: 12, Asszociativitás: bal] Jobbra „shift”-elés.
 $+$ [Precedencia: 13, Asszociativitás: bal] Összeadás.
 $-$ [Precedencia: 13, Asszociativitás: bal] Kivonás.
 $*$ [Precedencia: 14, Asszociativitás: bal] Szorzás.
 $/$ [Precedencia: 14, Asszociativitás: bal] Osztás.
 $\%$ [Precedencia: 14, Asszociativitás: bal] Maradékképzés.

1.4.1.1. Lusta operátorok

A lusta kiértékelés azt jelenti, hogy a két operandus közül először kiértékeli az elsőt (ami egy logikai/**bool** típusú kifejezés lehet csak), de az eredmény kiszámításához nem feltétlenül van szükség a második operandul kiértékeléséhez. Az eredményt a következő táblázat szemlélteti.

Bal operandus értéke	Lusta „és”	Lusta „vagy”
true	Jobb operandus értéke	true
false	false	Bal operandus értéke

A lusta operátor hátránya, hogy ugrást tartalmaz, ami lassíthatja a program végrehajtását. De vannak olyan esetek is, amikor a lusta operátor alkalmazása a futás időt mégis csökkenti (*ld. A.9. fejezet*).

lusta „és” [Precedencia: 4, Asszociativitás: bal]

```
function "&&"(bool, bool) : bool;  
function "&&>"(bool, bool) : bool;
```

Az operátor jelének két alakja ekvivalens, a fordítóprogram nem tesz különbséget a kettő között.

lusta „vagy” [Precedencia: 3, Asszociativitás: bal]

```
function "||"(bool, bool) : bool;  
function "||>"(bool, bool) : bool;
```

Az operátor jelének két alakja ekvivalens, a fordítóprogram nem tesz különbséget a kettő között.

1.4.2. Prefix operátorok

! [Precedencia: 15, Asszociativitás: jobb] Logikai negálás.

~ [Precedencia: 15, Asszociativitás: jobb] Bitenkénti negálás.

- [Precedencia: 0, Asszociativitás: jobb] Ellentett.

+ [Precedencia: 0, Asszociativitás: jobb] *Önmaga. A háttérben nem fog történni semmi, ez csak egy szintaktikai kiegészítés, hogy szimmetrikus legyen az „ellentett”-tel.*

1.5. Vezérlési szerkezetek

Vezérlési szerkezetet csak utasításblokkon belül lehet írni. Ez lehet a **main** blokk is, vagy egy új blokk. Függvény törzsének a blokkját nem lehet elnevezni.

1.5.1. Szekvencia

A szekvencia utasítások és (lokális változó) deklarációk¹⁷ sorozata. A forrás szövegben leírt sorrend szerint fogja a virtuális gép futtatni az utasításokhoz és deklarációkhoz tartozó kódot futtatni.

Utasításokat és változó deklarációkat vegyesen is lehet írni. Annyi a megkötés, hogy a lokális változót először deklarálni kell, és utána lehet használni. A változó deklarációkat `;` zárja (ld. 1.3.4. fejezet). Az utasítások többsége a csukó kapcsolószerű jellel (`}`) végződik – egyetlen kivétel a **do-while** ciklus (ld. 1.5.5.2. fejezet).

1.5.2. Blokk

Egy blokkon belül lehetőség van új blokk létrehozására. Az új blokkot opcionálisan el lehet nevezni, amire a belső blokkokban lehet hivatkozni.

```
...  
{  
    // utasítások a blokkon belül  
}  
...
```

A blokkban újra lehet definiálni a blokkon kívüli változóneveket, de – amennyiben a külső blokknak van neve, akkor – lehet hivatkozni a külső változókra is minősített névvel: **my_block_name:local_var**. Ebben a példában a **my_block_name** a blokk neve, és a **local_var** egy lokális változó a **my_block_name** nevű blokkban.

Egy példa található a fenti esethez a függelékben (ld. A.3. fejezet).

A minősített név használata tulajdonképpen csak egy blokkot választ ki a látható változók körének módosításához. Tehát ugyanúgy képes felfelé haladni a megfelelő változó kiválasztásához.

¹⁷Változók deklarációja és definíciója nincs megkülönböztetve. A két fogalom ugyanazt takarja, ahogy a C/C++ nyelvekben.

Ha a vezérlés elhagyja a blokkot, akkor a blokkban használt **string**-eket és rekordokat felszabadítja a virtuális gép.

1.5.3. Eljárás hívás

A nyelv nem definiálja explicit módon az eljárásokat úgy, mint az Ada. Mégis meg vannak különböztetve azok a függvények, amelyeknek nincs visszatérési értékük. Ezeket a függvényeket eljárásoknak tekinti a fordító. A megkülönböztetés oka csupán az, hogy az eljárásokat meg lehet hívni utasításként, míg függvényhívás csak kifejezésben szerepelhet (*ld. 1.6. fejezet*) – ebből kifolyólag a függvényhívás értékadásban is szerepelhet (*ld. 1.7. fejezet*).

Az eljárás meghívása a C nyelvben látott módon történik:

```
function Eljaras1(string s)
{
    ...
}

function Eljaras2()
{
    ...
}

main
{
    Eljaras1("parameter");
    Eljaras2();
}
```

Ha az eljárásnak nincs paramétere, akkor is kötelező kiírni a nyitó- és a csukó gömbölyű zárójelet.

1.5.3.1. Példa

Az alábbi program az eljárás hívást mutatja be. A **ketszer** eljárás a kapott paramétert kétszer írja ki a szabványos kimenetre.

```
#version 2.0

function ketszer(string s)
{
    write(s+s);
}

main
{
    ketszer("Kétszer írja ki.\n");
}
```

Kimenet:

```
Kétszer írja ki.
Kétszer írja ki.
```

1.5.4. Elágazás

A nyelvben az elágazás több módon is megadható. Az elágazás összes ágára igaz, hogy a nyitó- és csukó kapcsos zárójelet kötelező kitenni, és a blokkok el lehet nevezni. A feltételeknek minden esetben egyetlen logikai (**bool**) értéknek kell lennie – itt sincs automatikus típus konverzió.

Az egyes ágak blokkként viselkednek, tehát létre lehet hozni új lokális változót, működik az automatikus felszabadítás (*ld. 1.5.2. fejezet*).

Az általános konstrukció:

```
if(bool_expr1)
{
    // 1.
}
elif(bool_expr2)
{
    // 2.
}
elif(bool_expr3)
{
    // 3.
}
...
else
{
    // n.
}
```

A fenti kódban található ágak közül pontosan egy fog lefutni¹⁸. Ha az elágazás tartalmaz **elif**-et, és az elágazás feltétele hamis volt, akkor egyesével megvizsgálja, hogy melyik **elif** feltétele lesz igaz. Így halad addig, amíg van **elif**, és nem teljesül egy feltétel sem – amint teljesül egy feltétel, lefut az ághoz tartozó blokk, és vége az elágazásnak. Amennyiben egyetlen feltétel sem volt igaz, akkor az **else** ág fut le (ha van).

Ebből következik az elágazás egy fontos tulajdonsága. Hiába van több igaz értékű feltétel is, minden esetben az első igaz ághoz tartozó blokk lesz végrehajtva.

Az általános eset akárhány **elif**-et tartalmazhat. Az **elif** bevezetését az indokolta, hogy minden ág blokk. Ezért, ha többfelé kell el-

¹⁸Azért pontosan egy, mert minden esethez biztosan tartozik kód (**else** ág), és legfeljebb egy ág futhat le.

ágazni, akkor túl sok egymásba ágyazott blokk lenne, ami egy egyszerű konstrukcióval elkerülhető. A függelékben van egy példa arra, hogy miként nézne ki a fenti konstrukció (2 **elif**-fel) akkor, ha nem lenne **elif** (ld. A.4. fejezet).

Az **elif** módszertani szempontból is előnyösebb a klasszikus C-s elágazásnál – ez a csellengő **else** problémája. Egy megtévesztően tagolt C programról első ránézésre hibás következtetéseket lehet levonni. De az én elágazásomban minden ág csak blokk lehet, ami a csellengő **else** problémáját megszünteti, de felvet egy másik problémát, amit a függelék egyik példája is mutat: ha mégis többfelé kell elágazni, akkor nyelvi elemre van szükség, hogy átlátható legyen a kód (ld. A.4. fejezet).

Az általános konstrukcióból az **elif** és az **else** ágak is elhagyhatók. Minden az **elif**, mind az **else** opcionális, az elágazásban szereplő **elif**-ek száma tetszőleges lehet, az **else** ágak száma legfeljebb egy. A konstrukció legelső része, az **if** minden esetben kötelező, mert arról jön rá a fordító, hogy milyen kódot kell generálnia.

1.5.5. Ciklusok

A nyelv több ciklus konstrukciót is támogat. Ezek többsége a C nyelvben megszokott szintaxist használják, néhány megszorítással illetve módosítással. A ciklusmag csak blokk lehet, hogy a nyelv minimalizálja a tévedések számát (például: a meglévő ciklusmag egyetlen utasítást tartalmaz, és az új utasítás hozzáírásakor elmarad a nyitó- és csukó kapcsos zárójel).

1.5.5.1. while

A legegyszerűbb a **while** ciklus. A ciklus feltétele egyetlen logikai (**bool**) értékű kifejezés (ld. 1.6. fejezet). A ciklus a feltétel kiértékelésével indul, és ha a feltétel igaz, akkor végrehajtja a ciklusmagot, majd kezdődik minden előlről. Ha kezdetben a feltétel hamis volt, akkor a ciklusmag nem fog lefutni, vagyis a ciklusmagot addig futtatja le újra meg újra, amíg a feltétel igaz.

```
while(bool_cond)
{
    // ...
}
```

1.5.5.2. do-while

A **while** ciklushoz hasonló konstrukció a **do-while** (hátraltesztelés) ciklus. Ez annyiban tér el a fent ismertetett **while** ciklustól, hogy a ciklusmag legalább egyszer le fog futni. A ciklusmagot addig ismétli, amíg a ciklusfeltétel igaz – azzal a kiegészítéssel, hogy a ciklus végrehajtása nem a feltétel kiértékelésével kezdődik, hanem a ciklusmag lefuttatásával. A feltételnek egyetlen logikai értékű (**bool**) kifejezésnek kell lennie. Ez a C nyelvből ismert hátultesztelési ciklus.

```
do
{
    // ...
}
while(bool_cond);
```

1.5.5.3. for

A harmadik ciklus konstrukció a **for** ciklus. A C nyelvből ismert változattal ellentétben a kezdő értékadás legfeljebb egy (szimultán) értékadást tartalmazhat – ugyanez igaz a léptető utasításra is. A feltételnek egy logikai típusú kifejezésnek kell lennie – de ez elhagyható, és ebben az esetben a ciklusfeltétel az igaz (**true**) lesz¹⁹.

¹⁹A nyelvben a „szabályos” végtelen ciklus a **for(;;) { }**.

```

for(<assign_1>;<bool_cond>;<assign_1>)
{
    // ...
}

```

Mind az **<assign_1>**, mind az **<assign_2>** értékadás opcionális, egyiket sem kötelező megadni. Elhagyásuk esetén nem készül helyettesítő kód.

1.5.5.4. foreach

A nyelv legösszetettebb ciklusa a **foreach**. Jelenleg a **foreach** diszkrét intervallumon képes végigiterálni.

```

//      1 2 3 4 5 6              7
foreach(i in lo .. hi [inclusive/exclusive] [reverse])
{
    // ...
}

```

A **foreach** szerkezete:

1. Egy azonosítót kell megadni, amit a fordító létre fog hozni (ha létezik már olyan nevű, akkor a „régi” változóra minősített névvel lehet hivatkozni). A megadott azonosító lesz a ciklusváltozó. A ciklusváltozót – ellentétben a többi ciklussal – nem lehet megváltoztatni. A ciklusváltozó típusát a fordító határozza meg az intervallum típusa alapján.
2. Kulcsszó: **in**
3. Intervallum alsó határa.
4. Kulcsszó: **to** vagy **..**

5. Intervallum felső határa. Ennek a típusa meg kell egyezzen az alsó határ típusával. Ha a két típus egyezik, akkor a ciklusváltozó típusa is ez lesz. Elfogadott típusok: **i32**, **u32**, **i64**, **u64**. Ha a felső határ kisebb, mint az alsó, akkor a ciklusmag egyszer sem fog lefutni.
6. *Opcionális*. A felső határ módját állítja be. Ha az **inclusive** mód van kiválasztva, akkor a felső határ is benne lesz az iterált elemek között (alulról és felülről is zárt intervallum). Az **exclusive** mód esetén a felső határ nem lesz benne az iterált elemek között (alulról zárt és felülről nyílt intervallum) – ez a sorozat (**seq<T>**) elemének a bejárásakor hasznos²⁰. Az alapértelmezett az **inclusive**.
7. *Opcionális*. Ha a **reverse** meg van adva, akkor az intervallum bejárása fordított irányban történik. Ebben az esetben is érvényes az a kikötés, hogy ha a felső határ kisebb, mint az alsó határ, akkor a ciklusmag egyszer sem fut le. Tehát az alsó- és a felső határ jelentése következetes marad, és az, hogy az elemek bejárásának mi a sorrendje, az csak egy módosító lesz. Az alapértelmezett irány a kisebbtől a nagyobb felé.

A függelékben található egy példa a **foreach** és a sorozat (**seq<T>**) használatára (*ld. A.5. fejezet*).

1.5.6. Visszatérés függvényből

Eljárásból azonnali kilépést tesz lehetővé a fordító a **return;** utasítás használatával. Az összes lokális objektumot (rekordok, sorozatok, **string**) felszabadítja.

```
return;
```

Visszatérési értékkel rendelkező függvények esetében ennek a funkcionalitása ki van egészítve azzal, hogy a visszatérési értéket is a

²⁰A gond ott adódik, hogy ha a sorozat hossza 0, akkor ha kivonással lenne megoldva, akkor hibás intervallumot kapnánk.

return utasítással kell beállítani – továbbra is kilép a függvényből, de a visszatérési értékeket elhelyezi az adatveremben.

```
return value1, value2, ... valueN;
```

A **return** utasítás után egy (többértékű) kifejezésnek kell állnia. Optionálisan bezárójelezhető (ld. 1.6. fejezet).

1.6. Kifejezések

A kifejezés BNF-ben:

```
<expr> ::= <lit> | <lval> | "(" <expr> ")" |  
          <expr> <infix> <expr> |  
          <prefix> <expr> |  
          <ident> "(" <expr> ")" |  
          <stringlit> "(" <expr> ")" |  
          <expr> "," <expr>
```

<lit> Bármilyen literál lehet, kezdve a számkonstansoktól a szöveg literálig (ld. 2.1. fejezet).

<lval> Lokális változó, globális változó, formális változó, ciklusváltozó, minősített nevű lokális változó, balérték (ld. 1.7. fejezet).

<infix> Infix operátor (ld. 1.4.1. fejezet).

<infix> Prefix operátor (ld. 1.4.2. fejezet).

<ident> Egyszerű azonosító. Angol ábécé kis- és nagybetűi, számok, aláhúzás (nem kezdődhet számmal).

<stringlit> Szöveg literál.

Az utolsó alternatíva segítségével lehet többértékű kifejezést létrehozni. A nyelv értékadása is ezt a többértékű kifejezést használja, mert

ezzel van megoldva a szimultán értékadás is. Az ilyen kifejezések többértékűek, és a típusuk a benne szereplő kifejezések típusa²¹. A zárójellel az operátor precedenciát is felül lehet bírálni, de többértékű kifejezések létrehozásához is felhasználható. Erre mutat egy példát a függelékben található operátor túlterheléssel foglalkozó példája (*ld. A.6. fejezet*).

1.7. Értékadás

A nyelvben az értékadás nem kifejezés, mint például a C++ nyelvben, hanem utasítás, mint az Ada nyelvben. Az értékadás tervezésekor is fontos szempont volt a biztonság. Tipikus hiba C/C++ programokban, hogy az elágazás feltételében tévedésből nem egyenlőség vizsgálat van, hanem értékadás – ezt a fordító megengedi, és csak néhány fordító ad figyelmeztetést érte. Abban az esetben, ha az értékadás utasítás, akkor nem szerepelhet semmilyen feltételben, így ezt a problémát megoldotta ez a megszorítás.

Úgy tűnhet, hogy a láncolt értékadások előnye elvész. Erre a megoldás a szimultán értékadás. A nyelv megengedi, hogy egy többértékű kifejezést el lehessen tárolni változóba.

Egy fontos megszorítást tesz a nyelv a szimultán értékadásokkal kapcsolatban. A jobboldalon szereplő kifejezés (akár egyértékű, akár többértékű) minden értékét el kell tárolni. Tehát nem fordulhat elő olyan eset, hogy egy többértékű kifejezésnek csak az első néhány érték kerül eltárolásra, és a többi érték pedig elveszik (megsemmisül). Valamint egy kifejezés sem állhat magában, mint utasítás – ezt a C/C++ megengedi, de ez potenciális veszélyforrás.

A fordító sok ellenőrzést végrehajt az értékadások feldolgozásakor, és a sok megszorítás miatt több olyan értékadást is hibásnak ítél, ami például implicit cast-olást igényelne. E sok ellenőrzés célja, hogy a program biztonságosabb legyen, és ne legyenek meglepetések a futtatás során. A fordító igyekszik minél több ellenőrzést fordítási időben

²¹Nem szabad összetéveszteni a C++ nyelv szekvencia operátorával. Ebben a nyelvben a kifejezésnek több értéke és típusa lesz.

elvégezni.

Az értékadásnak két jele van, az egyik a C/C++ nyelvből ismert = jel, valamint az Ada és Pascal nyelvekből ismert := jel. A két jelölés ekvivalens, az elemző már nem is tesz különbséget a kettő között.

```
<lvalues> = <expr>;
```

A **<values>** balérték (*ld. 1.7.1. fejezet*), és a típusának pontosan meg kell egyeznie az **<expr>** típusával.

Az **expr** egy kifejezés (*ld. 1.6. fejezet*).

1.7.1. Balérték

A balérték olyan kifejezés, aminek értéket lehet adni: bármely, a nyelv által biztosított adattároló értékét jelentő kifejezés. Ez lehet egy lokális vagy globális változó, egy sorozat (tömb) egy eleme, illetve egy rekord mezője. Ez korlátozottabb, mint a C++ esetében, de minden, a nyelvben szereplő adatszerkezetnek értéket lehet adni. Minden balérték lehet jobbérték is.

```
<lvalues> ::= <lval> |  
             <lvalues> "," <lval>  
  
<lval> ::=   <ident> |  
             "global" ":" <ident> |  
             <ident> ":" <ident> |  
             <lval> "[" <expr> "]" |  
             <lval> "." <ident>
```

<ident> Balérték lehet egy lokális változó, vagy egy globális változó, ha az aktuális scope-ból látszik.

"global" ":" <ident> Globális változóra a **global** minősítéssel is lehet hivatkozni.

<ident> ":" <ident> Lokális változóra hivatkozás minősített névvel.

<lval> "[" <expr> "]" Egy indexelhető adatszerkezet indexelése²².

Az indexnek minden esetben egyetlen **u32** típusú kifejezésnek kell lennie.

<lval> "." <ident> Rekord mezőjéhez hozzáférés.

1.8. Kivételkezelés

A nyelvben függvényszintű kivételkezelés van. Egyrészt átláthatósági megfontolásokból²³, másrészt a virtuális gép egyszerűbben elkészíthető.

Mint a típusokat és a függvényeket is, a kivételeket is kell deklarálni (exportálni), vagy importálni (*ld. 1.3. fejezet*). A kivételnek nincs szigorú értelemben vett típusa, a fordítóprogram számára csak egy azonosító. Ez a program számára egy index lesz a lokális kivétel tömbben. A globális kivétel sorszámokat a virtuális gép fogja dinamikusan kiosztani a program betöltésekor (*ld. 3. fejezet*).

1.8.1. Kivételek kezelése

Minden függvény rendelkezhet kivételkezelő blokkokkal. Három különböző fajtát lehet megkülönböztetni:

1. Egzakt kivételhez tartozó kivételkezelő. Ez egyetlen, meghatározott kivételhez tartozik. Ebből a fajtából több is lehet.
2. Minden egyéb kivételhez tartozó kivételkezelő – amit az 1. pontban nem kezelt le. Ebből a fajtából csak egy lehet.

²²Indexelhető a **string** és a **seq<T>** (*ld. 2.1. fejezet*).

²³A forráskód sokkal átláthatóbb, ha nincs rengeteg egymásba ágyazott kivételkezelő blokk – ez leginkább a Java-ra jellemző.

3. Függvény kilépéskor végrehajtandó utasítások²⁴. Ebben a blokkban nem lehet **return** utasítás (*ld. 1.5.6. fejezet*). A **finally** blokk nem arra való, hogy visszatérési értéket állítson be a program! Az utasítások, amelyek ebben a blokkban vannak, mint esetben végrehajtásra kerülnek, ha a vezérlési szál elhagyja a függvényt – legyen az „normál” működés, vagy **return** utasítás, vagy kivétel. Ebből a fajtából csak egy lehet.

A kivételkezelés egy függvényen belül opcionális, egyik fajta sincs megkövetelve. Ezeket a blokkokat a függvény végén kell elhelyezni (ezek a blokkok után már nem lehet utasítás), a felsorolásban megadott sorrendben. A függvényeken kívül a **main** blokk is tartalmazhat kivételkezelő blokkokat²⁵.

Példa, amiben a **main** blokk ki van egészítve kivételkezelő blokkokkal.

```
#version 2.0

export
{
    exception MyException1;
    exception MyException2;
}

main
{
    exception(MyException1)
    {
        // 'MyException1' kivétel kezelése.
    }
}
```

²⁴Azért szerepel a kivételkezelés fejezetben a **finally**, mert befolyásolja a kivételkezelő mechanizmust a lefordított programban.

²⁵A **ctor** és a **dtor** jelenleg nem tartalmazhat kivételkezelő blokkokat.

```

    exception(MyException2)
    {
        // 'MyException2' kivétel kezelése.
    }

    exception(...)
    {
        // Minden olyan kivétel kezelése,
        // ami nem 'MyException1', és nem
        // 'MyException2'.
    }

    finally
    {
        // A main blokkból való kilépés előtt fut le.
    }
}

```

1.8.2. Kivételek kiváltása

Két lehetőség van kivétel kiváltására. A kettő között árnyalatnyi különbség van, de a gyakorlatban hasznosnak bizonyult.

1.8.2.1. A **raise**

Az egyik lehetőség a **raise**, ami az aktuális függvényből azonnal kilép, és a hívó függvény kivételkezelőjére ugrik – ha nincs a hívó függvényben kivételkezelő, akkor addig lép a hívó függvényre, amíg nem talál egy kivételkezelőt, vagy el nem fogy a hívási lánc. A kezeletlen kivételeket a futtató környezet kiírja a szabványos kimenetre.

A fenti példában látott **MyException1** kivétel felhasználásával (ld. 1.8.1. fejezet).

```
main
{
    raise(MyException1);
}
```

1.8.2.2. A **raise near**

A **raise near** egyetlen dologban különbözik a **raise**-től: nem lép ki az aktuális függvényből. Ettől eltekintve a kivétel terjedése megegyezik (ld. 1.8.2.1. fejezet).

A fenti példában látott **MyException1** kivétel felhasználásával (ld. 1.8.1. fejezet).

```
main
{
    raise near(MyException1);
}
```

A **raise near** nem egyetlen lexikális elem. Önmagában a **near** nem kulcsszó, csak a **raise** után írva lesz a **raise** és a **near** együtt kulcsszó. A **raise** és a **near** között tetszőleges számú fehér karakter („white space” – szóköz, tabulátor, újsor) lehet.

1.8.3. Szabványos kivételek

Az előre deklarált kivételeket a virtuális gép és a fordító is szabványos kivételeknek („*standard exceptions*”) hívja. Ezek a kivételek mindig be vannak töltve a virtuális gép kontextusába. Kiemelt szerepe van ezeknek a kivételeknek: a virtuális gép is kiválthatja ezeket a kivételeket, ezért van szükség arra, hogy előre legyenek deklarálva. A függelékben található egy példa, amely a gyakran előforduló szabványos kivételt bemutatja (ld. A.8. fejezet).

UnknownException Nem használt.

DivisionByZeroException Nullával való osztás történt.

OutOfMemoryException Memória foglalása nem sikerült, mert elfogyott a szabad memória. Ez **string**, rekord vagy sorozat foglалásakor fordulhat elő.

UnhandledControlPathException Úgy lépett ki egy függvényből, hogy nem volt visszatérési érték beállítva (nem volt **return**).

IndexOutOfRangeException Indexelhető objektum túlindexelése.

BadRecordFieldException Hibás rekord mező címezés. Ez a kivétel csak a fordító program fejlesztése során fordult elő. Jelenleg nem használt.

HandleDereferedToNullException Egy **nullhandle** dereferálva történt. Ez a beépített file- és könyvtárkezelő függvényeknél fordulhat elő.

ConstraintErrorException Megszorítás megszegése történt. Ezt a kivételt a konvertáló függvények válthatják ki.

InvalidPFNException Érvénytelen függvénymutató. Ez a kivétel hibás program fordításkor jelentkezhet.

InvariantViolationException *Jelenleg nem használt.* Invariáns megsértését jelzi ez a kivétel.

PreconditionViolationException *Jelenleg nem használt.* Előfeltétel megsértése esetén váltódik ki.

PostconditionViolationException *Jelenleg nem használt.* Utófeltétel megsértése esetén váltódik ki.

2. fejezet

Típusrendszer

A fordítóprogram további tevékenységei (a szintaktikai elemzésen kívül) a szemantikai ellenőrzések. Ide sorolható a típusok ellenőrzése, vagy annak ellenőrzése, hogy a **finally** blokkban van-e **return** utasítás. Ha egy programszöveg levezethető a nyelvtan segítségével, akkor még nem garantált, hogy szemantikusan helyes.

A fordító feladatai közé tartozik még az importálások és exportálások kezelése is, valamint a megfelelő assembly forrás létrehozása. Ehhez hozzátartozik az is, hogy a sorozatokhoz használt függvényeket is létrehozza¹ (ld. 2.1.2. fejezet).

2.1. Típusok

A nyelv legnagyobb erőssége a szigorú statikus típusrendszer². Egy „tisztá” program nem tudja megsérteni a statikus típusrendszert. Egy adott típusú kifejezésnek a típusa végleges, a típusát megváltoztatni csak explicit konvertáló függvényekkel lehet. A konvertáló függvény ellenőrzi, hogy a kért konverzió elvégezhető-e, és ha nem, akkor **ConstraintErrorException** kivételt vált ki (ld. 1.8.3. fejezet).

¹Ez a része a fordítóprogramnak valójában egy gyengén típusozott *template* példányosítás. Ez azt takarja, hogy nem feltétlen fog új kódot generálni a különböző típusokhoz tartozó kódokhoz.

²A jelenlegi verzió az altípusokat még nem támogatja.

A nyelvben előforduló konstans literáloknak is van típusa, de például a C++ nyelvvel ellentétben, egy konstans literál típusa egyértelmű(ld. 2.1.1. fejezet). Mint a kifejezések esetében, típus változtatása csak a megfelelő konvertáló függvény használatával lehetséges³.

Akkor „tisztá” („pure”) egy program, ha nincs felhasználó által hozzáadott C függvény hivatkozás. A későbbiekben elképzelhető egy olyan irányú fejlesztés is, hogy a virtuális gép figyelje az adatveremről kivett és berakott értékek típusát is, és hibás típuskezelés esetén kivételt vált ki.

Szintaktikai okokból kifolyólag egy **declare**, **import** és **export** blokkon belül nem lehetnek olyan típusok, amelyek hivatkoznak egymásra. Például a rekord esetén a mező típusa nem lehet ugyanabban a blokkban deklarált típus.

2.1.1. Alaptípusok

A nyelv biztosít néhány alaptípust. Ezeket a típusokat lehet használni változók típusaként, rekordok mezőinek típusaként, vagy lehet sorozatot készíteni.

i32 Előjeles, 32 bites egész szám, kettes komplementes ábrázolással.

Konstans szuffix: *nincs*. Példa konstans: **123**

u32 Előjel nélküli, 32 bites egész szám, kettes komplementes ábrázolással. Konstans szuffix: **u**. Példa konstans: **123u**

i64 Előjeles, 64 bites egész szám, kettes komplementes ábrázolással. Konstans szuffix: **l**. Példa konstans: **123l**

u64 Előjel nélküli, 64 bites egész szám, kettes komplementes ábrázolással. Konstans szuffix: **ul**. Példa konstans: **123ul**

f32 32 bites lebegőpontos szám, IEEE 754 szabvány szerint. Konstans szuffix: **f**. Példa konstans: **123.f**

³A konstans literál is kifejezés.

f64 64 bites lebegőpontos szám, IEEE 754 szabvány szerint. Konstans szuffix: *nincs*. Példa konstans: **123.0**

bool Logikai érték. Értéktartománya: {**true**, **false**}

char Karakter típus. 32 biten van ábrázolva, hogy a verem „*alignment*”-je ne sérüljön meg. Példa konstans: **'a'**

handle Extremális típus, 32 bites külső értékek (rendszerint mutatók) tárolására. Egyetlen értéke van a nyelvben: a **nullhandle()** függvény visszatérési értéke.

dhandle Extremális típus, 64 bites külső értékek (rendszerint mutatók) tárolására. Egyetlen értéke van a nyelvben: a **nulldhandle()** függvény visszatérési értéke.

string Karakterlánc (szöveg). Példa konstans: **"ez egy karakterlanc"**

A szuffixek lehetnek kis- és nagybetűsek is, a lexikális elemző elfogadja mindkettőt. Az összetett szuffixek lehetnek vegyesek is. Például: **uL** vagy **UL**.

Az egész típusok (**i32**, **u32**, **i64**, **u64**) konstansait több számrendszerben is meg lehet adni. A következő leírás csak a számrendszer leírását tartalmazza, az esetleges szuffixeket (**u**, **l**, **ul**) nem tartalmazza.

decimális 10-es alapú számrendszer. Ez az alapértelmezett számrendszer, nincs prefixe. Karakterek: {0, 1, ..., 9}. Példák: **1**, **27**.

hexadecimális 16-os alapú számrendszer. A konstansok prefixe: **0x** vagy **0X**. Karakterek: {0, 1, ..., 9, a, b, ..., f, A, B, ..., F}. Példák: **0x1**, **0x1B**.

oktális 8-as alapú számrendszer. A konstansok prefixe: **0o** vagy **00**⁴. Karakterek: {0, 1, ..., 7}. Példák: **0o1**, **0o33**.

⁴C/C++ nyelvekben az oktális számok prefixe a vezető 0. Ez megtévesztő is lehet. Ebben a nyelvben a vezető nullákat eldobja a fordító.

bináris 2-es alapú számrendszer⁵. A konstansok prefixe: **0b** vagy **0B**.
Karakterek: {0, 1}. Példák: **0b1**, **0b11011**.

A karakterlánc megadása állhat több literálból is. Ezeket a fordító összevonja, és egy karakterláncként tárolja el⁶. Ezzel a lehetőséggel lehet hosszú karakterláncokat is írni a kódba, mert megengedi a fordító, hogy több sorban legyenek az összevonásra kerülő literálok⁷. A karakterláncokat lehet indexelni (0-val kezdődik az indexelés) egy **u32** típusú kifejezéssel. Ez lehet jobb- és balérték is.

A felsorolt alaptípusok alapértelmezett paraméterátadása (a **string** kivételével) érték szerint történik. A **string**-ek esetében a konstans referencia az alapértelmezett, de ez módosítható (nem-konstans) referenciára a **mutable**⁸ kulcsszó segítségével.

Az alapértelmezett konstans referencia egy fordított gondolatot követ. Általában sokkal több az olyan paraméterátadás, ahol a paraméterül kapott objektumot nem változtatja meg a hívott függvény⁹. A statisztika szerint kevesebbszer kell kiírni a **mutable** kulcsszót, mint a **const** kulcsszót (például C++ esetében). Példaként fel lehet hozni a Java típusrendszerét, ahol nincs is lehetőség konstans paraméteradására.

Egy fontos érv a konstans referencia mellett az, hogy ha nincs kiírva a **mutable**, akkor hamarabb kiderül, hogy nem a megfelelő paraméterátadás lett kiválasztva, mert a fordító hibaüzenetben („error”) jelzi, hogy nem módosítható. Ha ugyanezt C++ nyelvben követjük el, akkor a fordító nem fog szólni, hogy nem módosítható, mert az alapértelmezett a nem-konstans referencia¹⁰. A probléma csak később fog kiderülni, amikor egy konstans objektumot kellene paraméterként átadni – de ezt már nem lehet, mert a formális paraméter típusa nem fogadja el a kons-

⁵Nagyon kevés nyelvben lehet írni bináris konstanst.

⁶Ez a funkció C/C++ nyelvekben is elérhető.

⁷Fehérszóközők (szóköz, újsor, tabulátor) elválaszthatják a literálokat.

⁸A függvény formális paraméterének típusa elé kell írni.

⁹Tapasztalataim szerint a függvényeknek jóval több, mint a fele konstans referenciát használ.

¹⁰Amennyiben a referencia szerinti paraméterátadást jelezzük a formális paraméterekben.

tans referenciát. Ezért sokkal biztonságosabbnak tartom az explicit jelzést a fordító számára, hogy nem-konstans referenciát kell használni. A nyelv egyik alapelve a biztonságosság – a statikus típusrendszer is a legnagyobb biztonságosságra törekszik.

2.1.1.1. Speciális szövegek

A fordító képes összevonni, és egy karaktersorozattá konkatenálni a fehér szóközökkel elválasztott karaktersorozatokat. Ez a funkció a legtöbb esetben használható, de vannak olyan esetek is, ahol nem használható – ilyen például a függvény nevének megadása szöveg alakban¹¹. A szöveg összevonást a C és a C++ nyelvek is támogatják.

A szöveg összeolvasztó mechanizmus kiválóan alkalmazható a fordító által automatikusa behelyettesített szövegek esetén. Lehetőség van az aktuális forrás file nevének behelyettesítésére, az aktuális sor beillesztésére szöveg alakban.

___**FILE**___ A file nevére helyettesítődik.

___**LINE**___ Az aktuális sor szöveg alakban. Ezt konvertáló művelettel számmá lehet alakítani (például a **function tou32(string s) : u32;** függvénnyel.

___**DATE**___ A rendszerből kiolvasott dátum **YYYY.MM.DD.** formátumban, ahol **YYYY** az év, **MM** a nap, és **DD** a nap decimális, vezető nullával kiegészített alakja.

___**TIME**___ Aktuális időpont **HH:MM:SS** formában, ahol **HH** az óra, **MM** a perc és **SS** a másodperc decimális, vezető nullával kiegészített alakja.

___**DATETIME**___ A ___**DATE**___ " " ___**TIME**___ rövidítése.

¹¹A függvény neve " jelek közé kerül.

2.1.2. Sorozatok

A sorozatok speciális típusok a nyelvben, mert ezek a típusok nem használhatóak sorozat alaptípusaként. Ez a virtuális gép típusreprezentációjából következik, mert a sorozatképzést egy módosító „*flag*”-gel jelöli a típuson belül. Ennek a problémának a megoldása a következő fejlesztési szakaszban implementálandó típuskonstrukció lesz¹².

A sorozat típus megadása:

```
<seq_type> ::= "seq" "<" <type> ">"
```

Ahol a **<type>** lehet alaptípus és rekord. A **<seq_type>** egy sorozat típus lesz.

A sorozatok és műveleteik a fordítóprogram számára *template*-nek tekinthető, mert csak azoknak a műveleteknek a kódját készíti el, amit a program felhasznál.

A programozás módszertanból ismert sorozat műveletek itt is megtalálhatóak. Egy művelet kivételével a nevük és paraméterezésük is megegyezik. A **dom** név helyett a **seqlen** nevet kell használni¹³. A **string**-ek mintájára a használható műveleteket tartalmazzák a rész-sorozat kiválasztását és a sorozat feltöltését is. A sorozatok belső elemeihez is hozzá lehet férni a **[]** segítségével: indexelni lehet egy **u32** típusú kifejezéssel. Ez lehet jobb- és balérték is. Azért lehet indexelni a sorozatokat, mert nincs külön tömb típus a nyelvben. Az indexelés 0-val kezdődik (úgy, mint a C/C++ nyelvek esetén).

A sorozatokhoz biztosított *template* függvények:

loext function loext(seq<T> s, T e) : seq<T>; A sorozat elejéhez egy új elem hozzáadása. (Egy új sorozatot hoz létre!)

hiext function hiext(seq<T> s, T e) : seq<T>; A sorozat végéhez egy új elem hozzáadása. (Egy új sorozatot hoz létre!)

¹²A konstrukció lényege, hogy egy létező típushoz be lehet vezetni egy álnevet, amit a fordító automatikusan felhasznál. A tervek szerint a **type** kulcsszó vezeti majd be.

¹³A **strlen** függvény terminológiája szerint.

lorem function lorem(seq<T> s) : seq<T>; Kitöröl a sorozat elejéről egy elemet. Üres sorozathoz üres sorozatot rendel. (Egy új sorozatot hoz létre!)

hirem function hirem(seq<T> s) : seq<T>; Kitöröl a sorozat végéről egy elemet. Üres sorozathoz üres sorozatot rendel. (Egy új sorozatot hoz létre!)

lov function lov(seq<T> s) : T; Visszaadja a sorozat első elemét. Üres sorozat esetén ***IndexOutOfRangeException*** kivételt vált ki.

hiv function hiv(seq<T> s) : T; Visszaadja a sorozat első utolsó. Üres sorozat esetén ***IndexOutOfRangeException*** kivételt vált ki.

lopop function lopop(seq<T> s) : seq<T>, T; Levesz a sorozat elejéről egy elemet, és visszaadja az rövidebb sorozatot és a törölt elemet. (Egy új sorozatot hoz létre!) Üres sorozat esetén ***IndexOutOfRangeException*** kivételt vált ki.

hipop function hipop(seq<T> s) : seq<T>, T; Levesz a sorozat végéről egy elemet, és visszaadja az rövidebb sorozatot és a törölt elemet. (Egy új sorozatot hoz létre!) Üres sorozat esetén ***IndexOutOfRangeException*** kivételt vált ki.

seqlen function seqlen(seq<T> s) : u32; Visszatérési értéke a sorozat elemeinek a száma.

subseq function subseq(seq<T> s, u32 o, u32 c) : seq<T>;
Létrehoz egy új sorozatot, ami a megadott sorozat részsorozatát fogja tartalmazni. Ha az intervallum kilóg, akkor azt vágni fogja („clamp”). Érvénytelen intervallum esetén üres sorozatot hoz létre. Az **o** paraméter az *offset*, vagyis a részsorozat első elemének indexe,

seqfill function seqfill(T e, u32 n) : seq<T>; Létrehoz egy *n* elemű, *T* alaptípusú sorozatot. Minden eleme *e* lesz.

A leírásban alkalmazott **seq<T>** típus az összes lehetséges *T* alaptípusú sorozatot jelöli.

A sorozatok alapértelmezett paraméterátadási módja a konstans referencia, de ezt a **mutable** kulcsszóval felül lehet bírálni, melynek hatására a sorozat elemei megváltoztathatók lesznek, de a sorozatot nem lehet felülírni (így a sorozat hossza sem változhat meg).

2.1.3. Rekordok

A deklarált, importált vagy exportált rekordok is típusok. A fordító biztosít minden rekordhoz másoló, inicializáló és törlő függvényt¹⁴. Rekord mező típusa akármilyen típus lehet, amit a nyelvben megtalálható. A rekordok megadásának módjával a szintaxis fejezet ehhez tartozó alfejezete foglalkozik (*ld. 1.3.3. fejezet*).

Minden rekord típus használható sorozat alaptípusaként, ezért minden rekordból lehet sorozatot létrehozni: **seq<rekord_neve>** (*ld. 2.1.2. fejezet*).

A rekordok alapértelmezett paraméterátadása konstans referencia szerint történik. A paraméterátadás módosítható (nem-konstans) referenciára a **mutable** kulcsszó segítségével. Ez csak a rekord mezőire vonatkozik, a rekordot lecserélni nem lehet (nem lehet a rekordnak értéket adni).

2.2. Típusok levezetése

A fordító legfontosabb tevékenysége a statikus típusellenőrzés. Ehhez szükség van egy mechanizmusra, ami minden kifejezés típusát¹⁵ meghatározza. A működéséhez ismerni kell a kifejezések konstrukcióit.

¹⁴Egyik függvény működését sem lehet megváltoztatni.

¹⁵A forrásszöveg értelmezésével párhuzamosan, típusozott részkifejezések alapján.

Ezek ismeretében a kifejezések típusa valóban mechanikus módon levezethető.

2.2.1. Balértékek típusa

Az értékadásban használható balérték típusa az a típus, amelyen típusú kifejezést el tud tárolni (*ld. 1.7.1. fejezet*). Ugyanez lesz a típusa, ha kifejezésben szerepel.

A **<lval>** típusa¹⁶:

<ident> A lokális változó típusa: **T**

```
<<type>> := T
```

"global" ":" <ident> A globális változó típusa: **T**

```
<<type>> := T
```

<ident> ":" <ident> A lokális változó típusa.: **T**

```
<<type>> := T
```

<lval> "[" <expr> "]" Sorozat esetén a sorozat alaptípusa, **string** esetén **char**.

```
<<type>> := if T is string then char
           elif T like seq<S> then S
           else <<error>>
           endif
```

<lval> "." <ident> A **<lval>** típusa: **T**

```
<<type>> := if T is record then select(T, <ident>)
           else <<error>>
           endif
```

¹⁶A balértékek szintaxisát leíró alfejezet jelöléseit használva (*ld. 1.7.1. fejezet*).

A **select** függvény a rekord típus megadott mezőjének típusát adja vissza, ha a mezőnév létezik a rekordban. Ha nem létezik a mezőnév, akkor **«error»** lesz a helyettesítési érték.

A **<lvalues>** típusa:

<lval> A balérték típusa: **T**

<<type>> := T

<lvalues> **" , "** **<lval>** A **<lvalues>** típusa: **Ts**, a **<lval>** típusa: **T**

<<type>> := Ts, T

A **,** a konkatenálást jelenti.

2.2.2. Jobbértékek típusa

Minden balérték használható jobbértékként is (vagyis használható kifejezésekben). A szintaxisban használt operátorok a típus levezetésben nem szerepelnek, mert az operátorok speciális szintaxissal is leírható függvények. Minden operátorhívás transzformálható függvényhívássá (a fordító is így dolgozza fel az operátorokat), ezért nincs részletezve a levezetésben. A leírásban csak az általános (**string** alakú) függvényhívás szerepel, mert a speciális (azonosító alapú) függvényhívás átírható az általános alakúvá.

Az **<expr>** típusa¹⁷:

<lit> A literál típusa (ld. 2.1.1. fejezet): **T**

<<type>> := T

¹⁷A kifejezések szintaxisát leíró alfejezet jelöléseit használva (ld. 1.6. fejezet).

<lval> A balérték típusa (*ld. 2.2.1. fejezet*): **T**

```
<<type>> := T
```

"(" <expr> ")" A kifejezés típusa (*ld. 1.6. fejezet*): **T** (A zárójelezés nem változtatja meg a kifejezés típusát.)

```
<<type>> := T
```

<stringlit> "(" <expr> ")" A kifejezés típusa: **T**

```
<<type>> := select_overload(<stringlit>, T)
```

A **select_overload** függvény kiválasztja azt a **<stringlit>** nevű függvények közül azt a túlterhelt változatot, ahol a függvény paramétere pontosan megegyezik **T**-vel – ha létezik ilyen függvény, akkor a helyettesítési értéke a függvény visszatérési értékének a típusa, egyébként **«error»**.

<expr> ", " <expr> Az első **<expr>** típusa: **T1**, a második **<expr>** típusa: **T2**

```
<<type>> := T1, T2
```

A **,** a konkatenálást jelenti.

3. fejezet

Virtuális gép

A fordítóprogram a virtuális géphez (*ld. 3.1. fejezet*) készíti el a kimenetet egy assembly forrás formájában. Ezt az assembly forrást a virtuális gép assemblere fordítja futtatható kóddá (*ld. 3.2. fejezet*). Az így előállított bináris állományt (a kiterjesztése **.skb** (*ld. 3.3. fejezet*)) a virtuális gép futtatja (*ld. 3.4. fejezet*).

A virtuális gép kezeli a programokhoz rendelt memóriát, amit a program használhat. Szükség esetén foglal még a program számára – ha már elfogyott a hozzárendelhető memória, akkor a memóriefoglalást tartalmazó utasítás **OutOfMemoryException** kivételt vált ki (*ld. 1.8.3. fejezet*). A program felszabadítása során a programhoz hozzárendelt memóriát felszabadítja a virtuális gép, ami más programok számára felhasználhatóvá válik.

3.1. A virtuális gép felépítése

A virtuális gép – amin a fordítóprogram által lefordított kód fut – két veremmel dolgozik. A lefordított kód futása során egy verembe fog számokat berakni, számokat manipulál (összead, kivon, stb.) és számokat vesz ki.

A virtuális gép által végrehajtható utasítások listája a függelékben megtalálható (*ld. A.12. fejezet*).

3.1.1. Az adatverem

Az adatverembe („*datastack*”) a futás során kiszámított értékeket tárolja el. Ebben a veremben található a függvények paraméterei és visszatérési értékei is. Hatékonysági okokból a verembe csak 4 byte vagy annak többszöröse méretű adatokat rak a verembe a virtuális gép¹. A verem alapú működés következménye, hogy könnyen megoldható a szimultán értékadás, és a több visszatérési érték. Egy másik következmény az összes visszatérési érték felhasználásának kötelezővé tétele is – ez kényszerként jelentkezett, másrésről hasznos, és módszertanilag fontos, mert csökkenthető a nyelv sebezhetősége. Ez a megoldás elgépeléseket képes kiszűrni, valamint a helytelen függvény felhasználási módot is szűri.

A C és a C++ nyelvek a visszatérési értéket az **(E)AX** regiszterben adják vissza, amelynek az értékét „büntetlenül” el lehet dobni. A struktúra visszatérési értéket pedig a veremben tárolják, amit szintén el lehet dobni. A C++ a visszatérési értéként kapott objektumokat temporális objektumként könyveli el, amit automatikusan felszabadít.

3.1.2. A végrehajtási verem

A végrehajtási verembe („*callstack*”) kerülnek a lokális változókat tartalmazó táblázatok (függvényenként legfeljebb egy, „*Dynamic Variable Table*” (DVT)²), temporális objektumok címei a törlő függvénnyel (eljárással) együtt, valamint a hívások során eltárolt adatok, amelyek a függvényből való visszatéréshez szükségesek („*call*)*stack frame*”).

A temporális objektumok címeit és törlő függvényeit tároló adatszerkezet is egy verem, amelynek a méretét fordítási időben meg lehet határozni. Ezt a vermet „*CallStack Object Stack*”-nek hívja a virtuális gép (CSOS). Egy függvényenként legfeljebb egy ilyen hozható létre. Függvényből való kilépéskor a tartalmát kiüríti, és törli a virtuális gép. Ennek

¹Ez az oka annak, hogy a karaktereket is 32 biten tárolja.

²Azért dinamikus, mert a környezettől függően ugyanaz a tárterület egyik esetben lehet **u32**, másik esetben **string** vagy **f64** típusú is.

kapcsán kellett a virtuális gépet felkészíteni arra, hogy futtatás során képes legyen egy újabb futtatást lekezelni („*invoke*”).

A CSOS azért tekinthető veremnek, mert egy blokkokból való kilépéskor nem kell az összes objektumot törölni belőle, valamint egy blokkban nem kell a programnak tárolnia azt, hogy éppen hány elem van a veremben. A verem felépítést pedig a kivételek kezelése tette véglegesen szükségessé, mert a kivétel kiváltásakor esetleg ki kell lépni a függvényből, és túlságosan költséges meghatározni, vagy tárolni azt, hogy hány elemet és mit kell felszabadítani – azonban a verem felépítés mindent ki tudja szolgálni.

3.2. Az assembly nyelv

A magasszintű nyelv fordítóprogramjának kimenete a virtuális gépet vezérlő assembly nyelv forrása. Az assembly réteg legfontosabb feladata az ugrások feloldása. A futtathatóság érdekében ahol lehet, relatív ugrásokat használ a fordító. A virtuális gépnek van két abszolút címre ugró utasítása is. Az egyik csak a teljesség miatt létezik, a fordítóprogram nem használja: **jmpa**. A **jmpa** utasítás az ugrás címét az utasításba kódolt 24 bites adatban („*immediate data*”) tárolja³. A másik utasítás nem az utasításba kódolva tárolja a címet, hanem az adatveremről vesz le egy 32 bites értéket, és arra a címre ugrik: **jmpal**. Ezt az utasítást már lehet relatív ugráshoz is használni, mert a **pushpc** utasítás berakja az adatverembe a következő utasítás címét az adatverembe, és ehhez már lehet hozzáadva, illetve kivonni. Ennek akkor van szerepe, ha a relatív cím nem fér el 24 biten (az ugró utasítások 24 bitet tudnak felhasználni ugráshoz), vagy egy konkrét helyre kell majd ugornia a virtuális gépre (függvény hívás megspórolása).

Az assembly nem csak utasításokkal dolgozik, hanem biztosít néhány speciális szekciót, amelyben a virtuális gép számára nélkülözhetetlen információk tárolhatók. Ilyen például a metaadatok is, amit egy

³A cím utasításokban értendő, azaz meg kell szorozni 4-gyel, hogy byte cím legyen belőle.

programhoz lehet társítani. Azonban ezt a virtuális gépnek külön kell kezelnie, ezért nem lenne célszerű hozzárakni egy meglévő szekcióhoz. Továbbá ilyen szekció még az importálások és exportálások leírása is, ami az assembly nyelv szintjén is megjelenik. Két különböző utasítást használt a belső függvények⁴ hívásához (**enter**, **mkfpenter**) és az importált függvények hívásához (**dle**, **mkfpdle**) (ld. A.12. fejezet).

A legfontosabb információ, ami az assembly szintjén is megjelenik, az a függvény szignatúrák. Ez a túlterhelt függvények miatt létfontosságú, mert a dinamikus linkelés során a megfelelő a függvény megfelelő verzióját kell lefuttatnia a virtuális gépnek. Ez az információ a speciális szekciókban, és az utasítások között (a hívó utasítások⁵ paramétere egy olyan karaktersorozat, ami tartalmazza a függvény nevét, az argumentumok típusát és a visszatérési értékek típusát is.

3.3. A bináris állomány

A futtatható állomány tartalmazza az összes betöltéshez szükséges kiegészítő információt a program utasításain kívül. Ilyen információk az importált és exportált függvények, típusok és kivételek, a konstans táblázat, és a byte-sorrend cserélési leírás. Egy **skb** állomány lehet „little-endian”⁶ és „big-endian”⁷ byte-sorrendű, de a program betöltő mind a két formátumot képes betölteni – a kettő között az a különbség, hogy az egyiket⁸ gyorsabban tudja betölteni, mert nem kell a byte-sorrendet módosítani. Egy assemblerrel mind a két verziót elő lehet állítani (ld. 3.2. fejezet).

Az állomány fejléce (az állandó interpretációval rendelkező bitek) 12 byte méretű. Ebben megtalálható az **skb** azonosítója, a byte-sorrend azonosítója, és az egyes ofszet és méret mezők méretei.

⁴Nem importált függvények.

⁵Hívó utasítások: **enter**, **dle**. Függvény mutatót létrehozó utasítások: **mkfpenter**, **mkfpdle**.

⁶LSB: Least Significant Byte

⁷MSB: Most Significant Byte

⁸A natív byte-sorrendű verziót.

Az ötödik byte-ban tárolja a program belépési pontjának, az adat- és a végrehajtási verem méretének leírására használt byte-ok számát (1, 2 vagy 4 byte). Valamint a törzs „pad”-jét is itt tárolja.

A hatodik és hetedik byte az offset mezők méretét, a nyolcadik és kilencedik byte pedig a méret mezők méretét határozza meg (1, 2 vagy 4 byte – vagy 0, ha az adott táblázat nincs jelen). A következő táblázatok lehetnek egy bináris állományban:

- fejléc kiegészítés (jelenleg nem használt)
- metaadat táblázat
- konstans értékek táblázata
- byte-sorrend cserélési információk a konstans táblázathoz⁹
- mutatók a konstans táblázatban
- import tábla
- export tábla
- utasítások (ez az egyetlen kötelező mező)

A tizedik és tizenegyedik byte az egyéb belépési pontokat címének a méretét tartalmazza. Ezek:

- konstruktor
- destruktork

3.3.1. A konstans tábla

A bináris állományban elhelyezett konstans tábla tartalmazza azokat a konstansokat¹⁰ amelyeket a program betölthet az adatveremre, vagy új karaktersorozatot hozhat létre belőle.

⁹Byte-sorrendet 1 (*nincs csere*), 2, 4 és 8 byte-os mezőkön tud módosítani.

¹⁰Ezeket az értékeket a lefordított program nem változtatja meg.

Ebben a táblázatban nem csak karaktersorozatok (és az ezekhez tartozó mutatókat) tárolja, hanem 32 vagy 64 bites számokat (egész vagy lebegőpontos) is elhelyez a fordító. A 64 bites értékek többségénél a bináris állomány méretének csökkenése figyelhető meg. A kivételt azok a 64 bites értékek jelentik, amelyek leírhatók 24+1 bit segítségével – ugyanis ezeket képes egy speciális betöltő utasítással az adatveremre helyezni. A 24+1 bit valójában két különböző utasítást jelent, amelyek 24 bit „*immediate*” adatok képesek tárolni.

A konstans táblában tárolt értékeket memória másolással tölti be a virtuális gép, ezért fontos, hogy megfelelő sorrendben legyenek az értékek byte-jai. Ehhez a feladathoz kell mellékelni egy byte-sorrend cserélő táblázatot is, ami méret szerint rendezve tárolja a mezőket. Ez egy optimalizálás az assembler részéről, hogy a byte-sorrend cseréléshez felhasznált byte-ok száma minimális legyen.

3.3.2. A readskb program

A readskb programmal lehet egy **skb** tartalmát megnézni. Az **skb** állomány értelmezése során kilistázható a file fejléce, a metaadat táblázat, a byte-sorrend leíró tábla, és az import- és export tábla. Továbbá a virtuális gépen futtatható kód utasításainak értelmezését is képes a standard output-ra kiírni.

Lehet állítani az import- és export tábla részletezését (függvény szignatúrák, kivételek neveinek, és típusok megjelenítése). A függvények szignatúráiban a típusok nevei helyett a típus azonosítószáma szerepel (hexadecimális formában). A program elsősorban a fejlesztést segítő eszköznek készült, ezért nem volt szükség a típusok korrekt megjelenítésére, de a fejlesztési tervek között szerepel a típusok neveinek kiírása is.

Az utasítások megjelenítése 3 oszlopból áll.

1. Utasítás címe a program kezdőcíméhez képest (a cím utasításban van megadva). Hexadecimális szám.

2. Az utasítás, hexadecimális formában.
3. Az utasítás mnemonikja, és paramétere. A paraméter megjelenítéséhez használhatja a hexadecimális alakot (ekkor a **0x** előtag szerepel), és a decimális alakot is (előtag nélkül). Ha a paraméternek speciális jelentése van (például ugrások, belső mikroprogramok, és karaktersorozatok esetén), akkor **<** és **>** között található az értelmezés eredménye¹¹.

3.4. Futtatás

3.4.1. Program betöltés

A program betöltő feladata a bináris állomány (*ld. 3.3. fejezet*) értelmezése, és a benne található információk feldolgozása. Ha van, akkor a betöltött program konstruktorát meghívja, és ha hiba nélkül lefutott, akkor lesz a program betöltése sikeres.

A virtuális géphez tartozik egy állapotleíró adatszerkezet, amelyben többek között az importált és exportált függvények, kivételek és típusok. Ezeket a program betöltése során felhasználja, és bővíti. Importált függvények esetén teljes egyezést vár el a program betöltő, vagyis ha a függvény neve és paraméter típusai¹² megegyeznek, akkor a visszatérési értékeknek is egyezniük kell – ha nem egyeznek, akkor hibás a betölteni kívánt program. Rekordok esetén is teljes egyezésre van szükség, amit a név és a mezők típusainak ellenőrzésével valósít meg.

3.4.2. Betöltött program futtatása

Egy betöltött program természetes futtatása a **main** blokk utasításainak a végrehajtásából áll. A virtuális gép API-ja rendelkezik olyan

¹¹Relatív ugrások esetén az abszolút cím, belső mikroprogramok esetén a mikroprogram neve, karaktersorozatok esetén a karaktersorozat első 20 karaktere – amennyiben a karaktersorozat hosszabb, akkor azt a **>** előtt álló ... jelzi.

¹²Ez egyértelműen azonosítja a függvényt.

függvénnyel, amellyel a **main** blokkból készült utasítássorozatot lehet végrehajtani.

Egy program természetes futtatása a **main** blokk tartalmának futtatását jelenti. A mellékelt **skx** program (ld. 3.4.4. fejezet) is a **main** blokk tartalmát hajtja végre, és a végrehajtás után kilép – attól függetlenül, hogy sikeresen lefutott a **main** blokk, vagy volt kezeletlen kivétel.

3.4.2.1. Függvény meghívása API-n keresztül

Az API lehetőséget biztosít a program által exportált függvények végrehajtására is – megkerülve a **main** blokkot. Ez a lehetőség az eredeti felhasználási terület miatt került bele az API-ba¹³.

Az exportált függvény meghívása (felhasználása) 3 fontos lépésből áll.

1. A függvény „*handle*”-jének lekérése a virtuális gép környezetéből. Ezek az azonosítóval lehet hivatkozni a függvényre (például C nyelvből).
2. A függvény használata:
 - (a) Ha van, akkor a függvény paramétereinek elhelyezése az adatveremben. A paraméterek sorrendje természetes, vagyis az első paramétert kell először elhelyezni, majd a másodikat, stb.
 - (b) A függvény meghívása. Ez a virtuális gép meghívását eredményezi, és a függvény utasításait egy „*invoke*” környezetben hajtja végre. Az „*invoke*” környezet azt jelenti, hogy nem feltétlen üres végrehajtási veremmel („*call stack*”) kezdődik a végrehajtás, és a függvény lefutása után ugyanabban az állapotba kerül vissza a végrehajtási verem. Ennek akkor van

¹³A nyelv fejlesztése csak egy magnak indult, amit C-s függvényekkel lehet kiegészíteni, és azokat C-s programokból felhasználni.

jelentősége, amikor kölcsönös hívások vannak a végrehajtásban¹⁴.

- (c) Ha a függvénynek vannak visszatérési értékei, akkor azokat ki kell venni az adatveremből. Ebben az esetben az értékek kivételi sorrendje fordított, tehát először az utolsó visszatérési értéket kell kivenni, utána az utolsó előtti, stb.

- 3. A „*handle*” felszabadítása. Erre azért van szükség, mert a teljes programot csak akkor lehet felszabadítani, ha már semmi sem hivatkozik az exportált erőforrásaira (függvény, típus, kivétel).

3.4.3. Program felszabadítása

Egy program csak abban az esetben lehet felszabadítani, ha minden függvény „*handle*” (ld. 3.4.2.1. fejezet) fel lett szabadítva, és a program által exportált függvények, típusok és kivételek nincsenek használatban. Ez akkor teljesül, ha a programra nem támaszkodik másik program, vagy a ráépülő programok már fel lettek szabadítva.

Ha a programot fel lehet szabadítani, akkor meghívja a destruktort a virtuális gép, és utána fogja a program által exportált erőforrásokat felszabadítani, valamint felszabadítja a program által használt memóriát.

3.4.4. Az *skx* program

A fordítóprogram melléklete az *skx* program is, ami egy pehelysúlyú futtatókörnyezet. Paraméterben kapja a futtatandó programot, és a futtatáshoz szükséges további programokat („*libraries*”).

Annak a meghatározása, hogy milyen további programokra van szüksége a futtatni kívánt programnak, azt a felhasználónak (programozónak) kell tudnia, és ezeket parancssori argumentumként megadni. En-

¹⁴Például: C hívja a virtuális gépet, majd az visszahívja a C-t, ami egy exportált függvényt fog hívni, ami ugyanabban a virtuális gépben fog lefutni, de már egy „*invoke*” környezetben.

nek az az előnye, hogy egy programot nem feltétlen csak ugyanazzal a „library”-vel lehet futtatni, hanem lehet cserélni azokat a programokat (és így azokat a függvényeket is), amelyre a futtatandó program épül. A dinamikus linkelésnek köszönhetően csak a függvények, kivételek és típusok meglétére van szükség.

3.4.4.1. A program paraméterei

A skx program egy parancssori alkalmazás, amelyet a parancssori argumentumokkal lehet vezérelni. Alapvető funkciója egy **skb** állományban található **main** blokk futtatása.

A **-lib** kapcsolónak egy paramétere van, ami a betöltendő program **skb** állományának elérési útvonala. Ezt a programot nem fogja futtatni (a konstruktor és a destruktor kivételével), de a program által exportált erőforrások felhasználhatóak lesznek („library”). Ebből a kapcsolóból több is lehet.

A **-builtin** kapcsoló hatására az skx program hozzáadja a virtuális függvényhez a két belső függvényét. Az egyik a verziószámot adja vissza: **function skx_version() : u32;** A másik a C-s **clock()** függvény visszatérési értékét adja vissza: **function skx_clock() : u32;**

A **-meta** kapcsolónak egy paramétere van, ami az **skb** metaadatainak egy kulcsa. Ha létezik a megadott kulcsú metaadat, akkor a kulcshoz tartozó értéket kiírja a standard kimenetre. Ebből a kapcsolóból több is lehet.

A **-stop** kapcsoló hatására az skx nem fogja futtatni a **main** blokkot, hanem kilép. Ezt a metaadatok kiírásához lehet felhasználni, amikor csak a programban tárolt metaadatok kiírása a cél.

4. fejezet

Konstrukciós szempontok

4.1. Ötletek és megvalósítások

Sok ötletet merítettem a C, a C++, és az Ada nyelvekből, valamint további ötletforrás a C# és a Java nyelvek. Nem utolsó sorban a módszertanból megismert műveleteket és jelölést¹ is felhasználtam.

4.1.1. Szintaxis

A számomra legszimpatikusabb szintaxiscsaládot választottam a szintaxis tervezésekor. Ez állt legközelebb az általam elképzelt funkciókhoz. A Pascal nyelvcsalád sajátossága, hogy a felhasznált kulcsszavak hosszabbak (például a `{}` helyett **begin end**), ami nagyobb forráskódot eredményez – továbbá véleményem szerint elrejt a lényegi részét a programnak. Egy másik ok, ami miatt nem a Pascal nyelvcsaládot választottam az az, hogy a változók és az azonosítók fordított sorrendben vannak a forrásban. Ez néhány esetben félreértéseket okozhat.

4.1.1.1. Belépési pont

A C nyelv szintaxisát használtam fel a saját szintaxis tervezésekor. Apró különbségek felfedezhetők, például a **main** blokknak nincs para-

¹Például a szimultán értékadást és a többértékű kifejezések jelölését.

métere (C-ben ez lenne: **int argc, char* argv[], char* env[]**), és visszatérési értéke sincs (ami C-ben **int** lenne). A függvényhívás szintaktikája is a C nyelvből származik.

A jól elhatárolt belépési pont sokkal áttekinthetőbbé teszi a programot, mint például a C, C++ nyelvek esetében a **main**, de Ada nyelvben is egy speciális eljárás a belépési pont, aminek a neve megegyezik a forrásállománnyal.

Az olyan objektumorientált nyelvek, mint a C# vagy a Java, nem olyan elegánsan oldják meg a belépési pont definiálását, mint például a C++. Például Java-ban a belépési pontnak ugyan **main** a neve, és a paraméterezése is kötött, de ez a függvény egy osztály statikus metódusa kell, hogy legyen. Ebből következik, hogy minden osztálynak lehet saját belépési pontja. Az ebből adódó nemegyértelműséget úgy oldják meg, hogy futtatáskor ki kell jelölni egy osztályt, amelynek a belépési pontját használni fogja. Ez a fajta megoldás a saját nyelv esetén szóba se jöhetett, mert jelenleg nincsenek osztályok.

4.1.1.2. Operátorok

Az operátor megadásának módját (prefix- és infix operátorokat idézőjelek közé kell írni) az Ada nyelv mintájára készítettem el. A szövegként leírt név miatt egyszerűbb a nyelvtan, jobban hasonlít a függvények megadásának módjára², és több karakterből áll, ami csökkenti el elgépelésből fakadó hibákat.

A funkcionális nyelvek többségében is idézőjelek vagy aposztrófok között kell megadni az infix/bináris (vagy prefix/unáris) operátorokat. A saját nyelv esetében azért írok prefix és infix operátorokat unáris és bináris helyett, mert lehet olyan prefix operátort írni, amelynek egynél több paramétere is lehet (amennyiben egy kifejezés a paraméter).

Más nyelvekben is létezik lusta és mohó logikai operátor („és” és „vagy”), így eldönthető a programozó, hogy melyiket szeretné használni. Többek között az Ada és a Erlang nyelv is felkínálja a lusta és

²A függvény nevének, mint azonosító, a szöveg az általánosabb alakja.

a mohó logikai operátorokat is. Más nyelvek, mint a C++ is, csak a lusta logikai operátorokat tartalmazzák. Mindkettőnek vannak előnyei és hátrányai is, melyeket alább foglaltam össze.

Előnyök: Ha a második operandus kiértékelés költséges művelet, akkor előfordulhat, hogy nem kell kiértékelni. A második operandus kiértékelés okozhat futásidejű hibát is, amit az első operandus tesztel.

Hátrányok: Ha a második operandust nem kell kiértékelni, akkor egy ugrás lesz a kódban, ami a kiértékelést kihagyja. Az ugrás hatására a végrehajtás egy pillanatra belassul, mert a processzor „cache”-ben más utasítások vannak. A cache újra töltése pedig időigényes³.

A függelék tartalmaz egy példaprogramot, amely a lusta operátorok használatát mutatja be (*ld. A.9. fejezet*).

Vannak olyan nyelvek is, amelyek nem támogatják az operátorok túlterhelését, mint például a Java: két azonos típusú érték összehasonlításához a **equals** függvényt kell alkalmazni. Azonban ez az **equals** függvény asszimmetrikus lesz, mert paraméternek lehet **null**-t adni, de egy **null** értékű objektumnak nem lehet meghívni az **equals** műveletét.

A funkcionális nyelvek sokkal gazdagabb lehetőségeket nyújtanak az operátortúlterhelés területén, mint a saját nyelvem, mert a funkcionális nyelvekben (például a Clean vagy a Haskell) nem csak előre definiált operátorokat lehet túlterhelni, hanem saját operátort is be lehet vezetni (például be lehet vezetni a **=/=**, vagy a **==>** operátort is), valamint az operátorok precedenciája nem előre rögzített, hanem az operátor létrehozásánál lehet megadni. Ezt a lehetőséget nem egyszerű feladat beépíteni a nyelvbe, mert az ehhez tartozó feldolgozó algoritmus nagyon összetett. A tervezés során úgy döntöttem, hogy nem éri meg ezt a lehetőséget beépíteni a nyelvbe.

³Nem virtuális gépen történő futtatás esetén még rosszabb a helyzet, mert ott a „pipeline”-t is meg kell még várnia a processzornak.

4.1.1.3. Statikus típusrendszer

Egyes nyelvek megengedik az előjeles és előjel nélküli számok összehasonlítását (pl.: C és C++), ami szélsőséges esetekben teljesen hibás működést von maga után. Akár egyenlőség vizsgálatról, akár kisebb/nagyobb operátorról van szó, az egész számok keverése rejtett hibákhoz vezet – azonban a fordító egy figyelmeztetés mellett elfogadja. Ezért döntöttem a statikus típusrendszer szigorítása mellett. E miatt az előjeles és előjel nélküli számok nem hasonlíthatók össze. De az operátor túlterhelés miatt mégis lehetséges egy olyan operátort készíteni, ami a kevert egészek összehasonlítását végzi el – a különbség ott adódik, hogy a speciális eseteket kötelező explicit lekezelni, mert a tényleges összehasonlításhoz konvertálni kell az egyiket, ami kivételt is ki tud váltani, ha a konverzió nem lehetséges. A mellékletben a kisebb művelet egy túlterhelt megvalósítása is megtalálható (*ld. A.6. fejezet*).

Más nyelvek ezt a problémát úgy kerülik el, hogy az alaptípusokból kihagyják az előjel nélküli egész számokat (pl.: Java). Azért nem ezt a megoldást választottam, mert előjeles számokat indexeléshez felhasználni nem tűnt jó ötletnek. Egyrészt azért, mert akkor minden esetben ellenőrizni kell, hogy az index negatív szám-e, valamint ebből kifolyólag az értékészlet megfelelődik. Számomra sokkal természetesebb, ha az index típusa előjel nélküli. Az első elem indexe a 0, vagyis a legkisebb előjel nélküli szám.

Az erős típusrendszer másik előnye, hogy feltételeknek nem lehet megadni például egész számot, vagy mutatót (C, C++), ami rejtett hibákhoz vagy nehezen átlátható kódhoz vezet. Rejtett hibát okozhat az, ha például egész szám szerepel a feltételben, de nem a nem-nullaság a feltétel, hanem az, hogy a szám negatív-e. Ha lemarad a **0>** az egész szám mellől, akkor azt a fordító elfogadja, de nem úgy fog viselkedni a program, ahogy kellene. Az ilyen problémák kivédhetők, ha a feltétel típusa csak logikai lehet.

A saját nyelvemben a feltétel (például az elágazás feltétele) csak logikai (**bool**) típusú lehet. Tehát csak olyan kifejezés írható, ami egyen-

lőség vizsgálat, vagy kisebb reláció, esetleg logikai típusú kifejezések konjunkciója vagy diszjunkciója (ami lehet mohó és lusta kiértékelésű is).

Vannak olyan szkriptnyelvek, ahol egyáltalán nem létezik statikus típusrendszer (például PHP), de egyes funkcionális nyelv is a dinamikus típusozás mellett döntött (például Erlang). A statikus típusrendszer hiánya nehézzé teszi a program helyességének bizonyítását, valamint a program karbantarthatóságát is csökkenti – a felhasználói típusok bevezetése a programozás egyik kulcsfontosságú eleme.

4.1.1.4. Kifejezések vs. értékadások

A nyelv tervezése során szétválasztottam az értékadást (*ld. 1.7. fejezet*) és a kifejezéseket (*ld. 1.6. fejezet*) annak érdekében, hogy ne lehessen keverni a két alapvetően különböző fogalmat. A figyelmemet a C/C++ jellegű nyelvek keltették fel, mert lehet értékadást írni ciklus feltételben is. Ennek nyilván van előnye is, hiszen optimálisabb kódot lehet generálni hozzá, illetve szintaktikailag is rövidebb (kevesebb karakterből áll). Azonban a „kényelemnek” ára is van, ugyanis ez a lehetőség egyben egy potenciális veszélyforrás is. A nem megfelelő operátor alkalmazása hibákhoz vezet (ha az **operator==** helyett az **operator=** kerül a vizsgálni kívánt értékek közé. Van helyettesítő nyelvi támogatás a C-ben is a feltételbe írt értékadás elkerülésére⁴, de azoknak az alkalmazása nem kötelező.

Ezért döntöttem amellett, hogy az értékadás és a kifejezés különüljön el, mert így nem lehet értékadást írni feltételbe, amivel a rejtett hibákat el lehet kerülni. Vagyis az értékadás nem kifejezés.

Más érv is alátámasztja azt, hogy biztonságosabb megoldás az értékadás és a kifejezés szétválasztása. Vegyünk egy C++ nyelvbeli esetet, ahol adott egy 1 paraméteres eljárás. Legyen a vizsgált kódrészlet a következő:

⁴Az elágazásba írt értékadás helyett egy szekvencia, a **while** feltételébe írt értékadás helyett pedig a **for** ciklus.

```
LoadFileContent("filename");
```

Ha a kódból véletlenül kitöröljük a **LoadFileContent** azonosítót, akkor a kód továbbra is fordul, esetleg a fordító egy figyelmeztetést ad. Azonban a program nem az elvárt viselkedést fogja nyújtani. A problémát nehezebb felderíteni, ha a paraméter nem literál, hanem egy kifejezés. Az ilyen jellegű hibákra a ScriptKernelnyelv fordítási hibát ad, miszerint a kifejezés értékei nem lettek felhasználva (értékadásban vagy függvény paraméterben).

4.1.2. Kivételek

A nyelvem alapjául szolgáló C nyelvben nincs kivételkezelés⁵, így a kivételkezelés szintaktikáját nem tudtam onnan meríteni.

A kivételkezelés mechanizmusát az Ada nyelv alapján készítettem el. Azonban az Ada nyelvben bármilyen blokk tartalmazhat kivételkezelő kódot, de a saját nyelvben csak függvényhez lehet kivételkezelő kódot csatolni. Azért alakult így, mert így lehetett a leghatékonyabban megvalósítani a virtuális gépben⁶. Ez a progrozó számára nem jelent nagy hátrányt. A kivételek elnevezése a „*CamelCase*” konvenciót követi, és minden kivétel azonosító utolsó szava a **Exception** – ezt az elnevezési konvenciót a C# és a Java nyelvek is betartják.

A saját nyelv függvény szintű kivételkezelést biztosít (az Eiffel nyelv mintájára). A C++ **try-catch** mechanizmusát a fordítóprogram és a virtuális gép egyszerűsítése miatt vettem el, mert olyan funkciók megvalósítását igényelték volna, amelyek a hatékonyság csökkenését vonták volna maguk után. Más érv is szól emellett: egy kezdő programozónak egyszerűbb az a felfogás, hogy ha egy függvényben kivétel történt, akkor azt a függvény végén elhelyezett kód kezeli. A C++ más szemszögből közelíti meg ezt a témát: a **try** blokkba a „védett” utasításokat kell

⁵Egyedül a **setjmp** és a **longjmp** által biztosított csekély lehetőség van.

⁶A kivételkezelésből adódó túlterhelést így lehetett a legjobban minimalizálni.

írni, és ha ezek végrehajtása során kivétel lépett fel, akkor valamelyik **catch** ág kezeli le a típustól függően.

Az Eiffel szemlélete más okból is előnyös: a kivételkezelés során dönthet úgy a programozó, hogy a kivétel okát elhárította, és utasíthatja a programot úgy, hogy „próbálja meg ismét” az eljárást végrehajtani. Más lehetősége is van a programozónak: alternatív végrehajtási stratégia arra az esetre, ha az első nem sikerül (kivétel lép fel).

4.1.3. A bináris formátum

A lefordított kód egy **skb** kiterjesztésű file-ba kerül (*ld. 3.3. fejezet*). A formátum jelenlegi állapota egy hosszú folyamat eredménye. Az aktuális formátumban csak annyi byte információ van eltárolva, amennyi feltétlenül szükséges (helytakarékossgot és betöltési sebességet is figyelembe véve). A formátumban található ofszet és méret értékek a lehető legkevesebb byte információt tárolják annak érdekében, hogy kis programok esetében ne legyen nagy az adminisztráció. Régebbi verzióban mindenhol 32 bites értékek voltak eltárolva, és csak natív byte-sorrendet támogatott.

A jelenlegi verzió rugalmasan bővíthető, vagyis egy régebbi virtuális gép képes úgy betölteni a bináris állományt, hogy csak akkor ad hibát, ha nem tud feloldani valamilyen hivatkozást. A régi bináris állományra ez nem igaz, mert az egy nagy fix fejléccel rendelkezett, és nem volt képes dinamikusan kezelni a táblázatokat – például ha bekerült volna egy új típusú bejegyzés az export táblázatba, akkor a betöltő program nem tudta volna értelmezni a táblázatot. Az új változatban lehetőség van az ismeretlen bejegyzések átugrására, és azoknak jelzése a felhasználó felé.

4.2. Algoritmusok

Ez a fejezet az absztrakt algoritmusok implementálásának lépéseit mutatják be. Az implementáció elkészítése három lépésből áll:

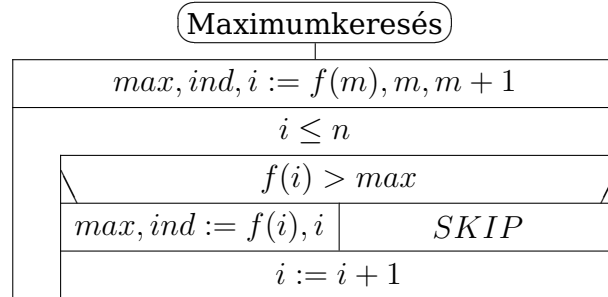
1. Absztrakt algoritmus kiválasztása.
2. Az absztrakt algoritmus specializálása a feladathoz.
3. Absztrakt algoritmus „átmásolása” a forráskódba⁷

4.2.1. Maximumkeresés

4.2.1.1. Absztrakt algoritmus

$$f : [m..n] \mapsto \mathbb{H}$$

$$A = (m : \mathbb{Z}, n : \mathbb{Z}, ind : \mathbb{Z}, max : \mathbb{H})$$

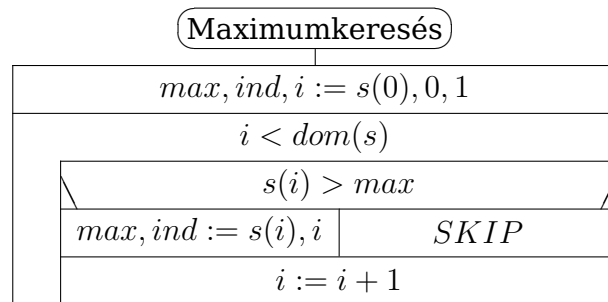


4.2.1.2. Specializált algoritmus

Speciálisabb eset, ha az f nem függvény, hanem egy sorozat. A példában a $\mathbb{H}$ halmazt helyettesítem a **f32** típussal, és a sorozatot a s jelöli. A sorozat indexei a $[0..dom(s) - 1]$ intervallum elemei.

$$A = (s : seq(\mathbf{f32}), ind : \mathbb{Z}, max : \mathbf{f32})$$

⁷A szükséges szintaktikai kiegészítések elkészítése után.



4.2.1.3. Implementáció

```
function maxker(seq<f32> s) : u32, f32
{
    u32 dom = seqlen(s);
    if(dom==0u) { raise(IndexOutOfRangeException); }

    f32 max;
    u32 ind, i;

    max, ind, i := s[0u], 0u, 1u; // [1]

    while(i<dom) // [2]
    {
        if(s[i]>max) // [3]
        {
            max, ind := s[i], i; // [4]
        }

        i := i + 1u; // [5]
    }

    return ind, max;
}
```

A kódban bejelölt sorok (// [1], stb.) sorszámai a struktogram adott

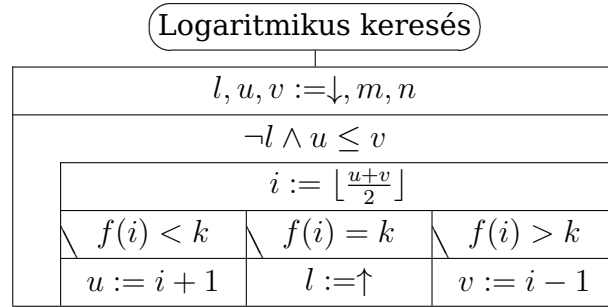
sorának az implementációja. Megfigyelhető, hogy a kód és a struktogram szintaktikusan is megfelel egymásnak. Jelölésbeli különbség a sorozat indexelése ((**i**) helyett [**i**]), és az index típusa (a struktogramban „sima” egész volt, a kódban pedig előjel nélküli egésznek kell lennie).

4.2.2. Logaritmikus keresés

4.2.2.1. Absztrakt algoritmus

$$f : [m..n] \mapsto \mathbb{H} \nearrow$$

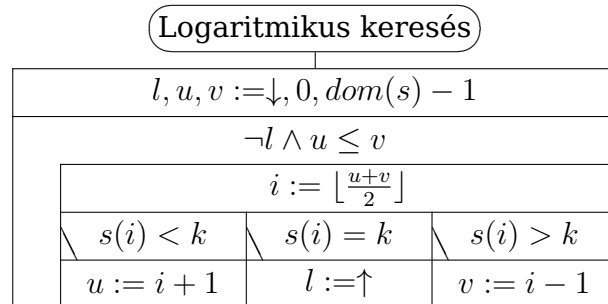
$$A = (m : \mathbb{Z}, n : \mathbb{Z}, k : \mathbb{H}, l : \mathbb{L}, ind : \mathbb{Z})$$



4.2.2.2. Specializált algoritmus

Az f függvény helyett egy **f32** típusú elemekből álló, növekvően rendezett sorozatot használlok. Így az n helyett 0, az m helyett $dom(s) - 1$ lesz az intervallum.

$$A = (s : seq(\mathbf{f32}), k : \mathbf{f32}, l : \mathbb{L}, ind : \mathbb{Z})$$



4.2.2.3. Implementáció

```
function logger(seq<f32> s, f32 k) : bool, u32
{
    i32 dom = toi32(seqLen(s));
    if(dom==0) { return false, 0.f; }

    bool l;
    i32 u, v, i;

    l, u, v = false, 0, dom-1; // [1]

    while(!l && u<=v) // [2]
    {
        i := (u+v)/2; // [3]
        f32 si := s[tou32(i)];

        if(si<k) // [4.1]
        {
            u := i + 1;
        }
        elif(si==k) // [4.2]
        {
            l := true;
        }
        else //elif(si>k) // [4.3]
        {
            v := i - 1;
        }
    }

    return l, tou32(i);
}
```

Az implementációban az index előjeles egész, mert a logaritmikus keresés akkor is véget érhet, ha a nagyobb index (**v**) kisebb, mint a kisebb index (**u**). A nagyobb index lehet negatív is, ha csak a nagyobb index változik – ezért kell az előjeles egész.

A **[2]** sor implementációja során ki kellett cserélni a műveleti jeleket a nyelv által elfogadott jelekre: $\neg$ helyett **!**, $\wedge$ helyett **&&**.

A **[3]** sorban alsó egész résznek kellene lennie, de a nyelv úgy definiálja az egészszel való osztást, hogy a tört részt eldobja, vagyis valóban alsó egész rész lesz. A következő sorban eltárolom a sorozat **i**-edik elemét egy lokális változóban, hogy ne kelljen többször hozzáférni a sorozathoz. Egy konvertálás is található (**tou32**), mert a sorozatot csak előjel nélküli egészszel lehet indexelni. Kivétel nem fog kiváltódni, mert az absztrakt algoritmus biztosítja azt, hogy a számolt index (**i**) mindig érvényes lesz.

A többágú elágazás (**[4]**) megvalósítása az **if-elif-else** vezérlési szerkezettel történt. Optimalizálási szándékból a harmadik ágon (**[4.3]**) nem történik feltételvizsgálat, mert a harmadik feltétel mindig igaz lesz, ha az első kettő hamis⁸.

4.3. Eredmények

4.3.1. A *Hello World* program

A *Hello World* programnak ScriptKernelnyelven nagyon rövid a forrása.

```
#version 2.0
main
{
    write("Hello World!\n");
}
```

⁸A nyelvben nem kötelező minden esetet feltétellel ellátni. Ha az utolsó eset adódik, akkor az **else** használata optimálisabb megoldást eredményez.

A lefordított program binárisa 100 byte, ami sokkal kisebb, mint egy hasonló funkcionalitású C vagy C++ programoknál. A Java osztály, amely kiírja a képernyőre a *Hello World!* szöveget, 108 byte.

A fenti forrással ekvivalens C program:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

Ennek a C forrásnak a mérete sokkal nagyobb, mint a ScriptKernel-nyelven írott változat.

4.3.2. Gyorsrendezés

A gyorsrendezés megvalósításában látható, hogy nincs szükség cserélő függvényre, mert a cserék elvégzésére nyelvi támogatás van. A megvalósítást egy C nyelven írt gyorsrendezéssel hasonlítottam össze.

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
swap(&arr[--l], &arr[beg]);
```

helyett

```
l = l - 1u;
arr[l], arr[beg] = arr[beg], arr[l];
```

Az ScriptKernelnyelvbeli megvalósítás előnye, hogy nincs benne függvényhívás, és a kód is sokkal olvasmányosabb, mint a C-s megvalósítás.

A ScriptKernelnyelven megírt gyorsrendezés az indexelés során ellenőrzi az indexhatárokat, és amennyiben hibás a függvény, vagy nem megfelelően lett meghívva (nem a „*frontend*” függvénnyel lett elindítva), akkor futásidejű hibát ad az indexhatárok megsértése miatt: ***IndexOutOfRangeException***.

A megvalósításban felhasználtam azt, hogy a paraméterül kapott tömb elemei megváltozhatnak (a tömb két elemét meg lehet cserélni), de a tömb mérete nem változhat (nem lehet lecserélni a tömböt). Ezt a **mutable** kulcsszóval lehet jelezni a fordító felé, amit a deklarációban helyeztem el:

```
function qsort(mutable seq<i32> arr, u32 beg, u32 end);
```

A gyorsrendezést megvalósító függvény egy rekurzív függvény, ezért el kellett helyezni a deklarációját a **declare** blokkban:

```
declare
{
    function qsort(mutable seq<i32> arr, u32 beg, u32 end);
}
```

A teljes kód megtalálható egy példával kiegészítve a függelékben (*ld. A.10. fejezet*).

4.3.3. Az eddigi legösszetettebb program: *WorldEditor 5*

Az általam ScriptKernelyelven írt programok közül a *WorldEditor 5* nevet viselő program a legösszetettebb és egyben legnagyobb. Több, mint 17900 utasítást tartalmaz a program, amit 114 KiB forráskódból fordított le. A forrás összesen 3300 sor (üres és megjegyzés sorok nélkül).

Ez a program egy 3-dimenziós pályaszerkesztő, egér és billentyűzet események, pálya leíró értelmezővel és belső szintér reprezentációval. A nyelv segítséget nyújtott a sorozat kezelő függvényekkel, mert a program statikusan típushelyes. Ami még nagy segítség volt, az az operátor túlterhelés, amivel a saját **Vector3** típushoz készítettem összeadó és kivonó műveleteket. A program nem tartalmaz sok kivételkezelést, mert a statikus típushelyesség miatt nincs rá szükség. A pályaszerkesztő biztosít néhány olyan funkciót is, amit csak nem konstans rekordokkal (a szintér elemeit rekordokban tárolom) lehet elvégezni. A **mutable** módosítót a rekord paraméterű függvényeknek kevesebb, mint a felében kellett alkalmazni (*ld. 2.1. fejezet*).

5. fejezet

Összegzés

Dolgozatomban ismertettem a ScriptKernel nevű nyelv szintaxisát és szemantikáját. Bemutattam, hogy milyen előnyökkel jár a szigorított szintaxis, és az erős statikus típusrendszer. A fontos nyelvi elemeket részletesen ismertettem, példákkal egészítettem ki. A nyelvhez tartozó fordítóprogramot és futtató környezetet (virtuális gép) ingyenesen le lehet tölteni (*ld. 5.1. fejezet*).

A nyelv szigorú szintaxisa és szemantikája minimalizálja a véletlen hibákat, és növeli a program karbantarthatóságát. Az elkészített szintaxis illeszkedik az ELTE-n tanított programozási módszertanhoz, ezért oktatásban is felhasználható a nyelv.

A fejlesztéshez több tesztet is készítettem, melyeket a regressziós tesztelés során is használok. Ezek egy része a letölthető csomagban is megtalálható.

A fordítóprogram, a virtuális gép, valamint az egyéb eszközök is C++ nyelven készültek – a virtuális gépben felhasznált saját memória allokátor pedig C nyelven. A felhasználhatóság érdekében C interfésszel láttam el a komponenseket, így könnyen beépíthető C illetve C++ programokba. A virtuális gép betöltő mechanizmusa felhasználható az online kódcsereére „*hot codeswap*”. A virtuális gép környezetéhez hozzá lehet adni C és C++¹.

¹Abban az esetben, ha a C++ függvény C-s interfésszel rendelkezik: **extern "C"**.

A nyelvhez készítettem még egy „*gedit*” szerkesztőben használható szintaxis kiemelőt is, amivel a fejlesztés gördülékenyebben zajlik, mert a kulcsszavakat és lexikális elemeket kiemeli. Ha elírás történik, és ha az adott helyen kulcsszónak kell állnia, az még a szerkesztőben kiderül, és nem a fordító hibaüzenetei között értesül róla a programozó. A szintaxis kiemelő hasznos eszköz egy program fejlesztése során.

A használhatóságot pedig az eddigi legnagyobb programom támasztja alá: 3300 sornyi forráskódból 17900 végrehajtható utasítás fordult le. A program egy 3-dimenziós pályaszerkesztő, amit a bevezetőben említett autós játékhoz is fel lehet használni.

5.1. Elérhetőség

A nyelv honlapja: <https://people.inf.elte.hu/baaqaai/skc>

A „*gedit*” szerkesztőhöz letölthető a nyelv szintaxis kiemeléséhez használható állomány²: <https://people.inf.elte.hu/baaqaai/skc/sk.lang>

A fordító program és a futtató környezet elérhető a https://people.inf.elte.hu/baaqaai/skc/skc_0799.html címen.

²Az állományt a `/usr/share/gtksourceview-3.0/language-specs` könyvtárban kell elhelyezni.

5.2. Továbbfejlesztési lehetőségek

5.2.1. Elő- és utófeltételek

A nyelv következő kiegészítése az elő- és utófeltétel ellenőrzés lesz. Ezt minden függvényhez meg lehet adni, és a végrehajtás során ellenőrzésre kerül – kérésre az ellenőrzések elhagyhatóak. Ez a funkció főleg a program fejlesztése során hasznos. A szintaxisa a következő lesz:

```
function f(i32 a) : bool
pre { a>0 }
post(b) { a0 = a; b==(a0%2) }
{
    return a%2;
}
```

5.2.2. Referencia szerinti paraméterátadás

Ennek a funkciónak az a célja, hogy úgy lehessen (akár rekordokat és sorozatokat is) értékeket paraméterként átadni, hogy azokat le lehessen cserélni. Ez a módosító kizárólag függvény paraméter módosítója lehet (ahogy a **mutable** is). Az új kulcsszó: **ref**

5.2.3. Új foreach

A **foreach**-nek lesz egy új változata is, amely nem intervallumon, hanem egy sorozaton (**seq<T>** vagy **string**) képes végigiterálni. Az iterátor az aktuális elem referenciája lesz. A referencia az lehet **mutable** referencia is, ha a sorozat **mutable**.

5.2.4. Altípusok

Ez a változtatás nagyobb horderejű, és pontosan ki kell dolgozni, hogy a statikus típusrendszer biztonsága ne legyen kizárható. Terveim

szerint két különböző módon lehet majd új típusokat származtatni már meglévőkből:

1. Álnév („*alias*”) bevezetése. Az eredeti és az új típus ekvivalens, de több névvel lehet hivatkozni ugyanarra a típusra³.
2. Altípus („*subtype*”) létrehozása. Az új típus egy teljesen különálló típus lesz, de lehet konvertálni (az új és az eredeti típus felépítése megegyezik, de az új típushoz tartozik egy invariáns is⁴.

5.2.5. Osztályok

A későbbiekben szeretném bevezetni az osztályokat is nyelvbe. El kell gondolkodni azon is, hogy a nyelv megengedje-e az altípusos polimorfizmust, valamint azon, hogy az öröklődés mentén legyen-e automatikus altípusosság, vagy külön kell azt jelezni.

Még csak ötleteim vannak, de a komplett koncepció még nem készült el.

5.2.6. Párhuzamosság, elosztottság

Távolabbi terveim között szerepel a párhuzamosság nyelvi szintű támogatása vagy a nyelv felkészítése arra, hogy elosztott rendszeren is képes legyen hibátűrő módon futni. Még nincs kidolgozva, hogy milyen módon lehessen megadni azokat a részleteket, amelyeket szét lehet osztani a különböző számítógépek között. Egyik lehetőség a feladatkörök („*task*”) bevezetése (amit akár többszálúan is végrehajthat), a másik lehetőség az, hogy a fordító program dönt.

³Ez annak kapcsán merült fel, hogy a jelenlegi lehetőségekkel nem lehet sorozatokba sorozatot rakni. Ez a típusok hordozható azonosítói miatt van így.

⁴A típushoz létre kell hozni egy invariáns függvényt is (**function invariant(T t) : bool;**), amit a fordító minden értékadáskor ellenőrizni fog.

Irodalomjegyzék

- [1] Nyékyné Gaizler Judit: *Programozási nyelvek*
Kiskapu Kiadó, 2003.
- [2] Dr. Csörnyei Zoltán: *Fordítóprogramok*
Typotex Kiadó, 2006.
- [3] Fóthi Ákos: *Bevezetés a programozáshoz*
ELTE Eötvös Kiadó, 2006.
- [4] Bjarne Stroustrup: *The Design and Evolution of C++*
Addison-Wesley Professional, 1994.
- [5] Ellis & Stroustrup: *The Annotated C++ Reference Manual*
Addison-Wesley, 1990.
- [6] C++ reference,
<http://www.cplusplus.com/reference>
2012. szeptember 16.
- [7] C# reference,
<http://msdn.microsoft.com/en-us/library/618ayhy6.aspx>
2012. szeptember 21.
- [8] Ada Reference Manual,
<http://www.ada-auth.org/arm.html>
2012. október 4.

- [9] Eiffel Documentation,
<http://docs.eiffel.com/book/method/quick-reference-eiffel-programming-language>
2012. szeptember 29.
- [10] The Clean Collection,
<http://www.cleanlanguage.co.uk>
2012. január 27.
- [11] The Haskell Programming Language,
<http://www.haskell.org/haskellwiki/Haskell>
2012. január 24.
- [12] UnrealScript,
http://wiki.beyondunreal.com/UnrealScript_overview
2012. november 17.
- [13] ERLANG, *<http://www.erlang.org>*
2012. november 11.
- [14] Java reference,
<http://www.oracle.com/technetwork/java/index-jsp-142903.html>
2012. december 9.
- [15] The programming language Lua,
<http://www.lua.org>
2012. február 5.
- [16] JavaScript Tutorial,
<http://www.w3schools.com/js/default.asp>
2012. február 20.
- [17] REXX, *<http://www.rexxinfo.org>*
2012. március 2.
- [18] PHP, *<http://php.net>*
2012. augusztus 11.

A. Függelék

A.1. A nyelv BNF leírása

A nyelvtan BNF leírása (nincs minden kifejtve, a részletek a megadott fejezetekben találhatóak). A BNF leírásban nem lehet szemléltetni a nyelv legfontosabb mechanizmusát, a statikus típusrendszert, ami a nyelv legerősebb eszköze a helyes programok elkészítéséhez.

```
<start> ::= <signature> <program_body>

<signature> ::= "#version" "2.0"

<program_body> ::= <<empty>>
                  | <block> <program_body>
                  | <option> <program_body>
                  | <meta> <program_body>

<block> ::= "main" <instrblock_exc>
          | "ctor" <instrblock>
          | "dtor" <instrblock>
          | "import" "{" <dynobjs> "}"
          | "export" "{" <dynobjs> "}"
          | "declare" "{" <dynobjs> "}"
          | "global" "{" <definitions> "}"
          | <funcsig> <instrblock_exc>

<instrblock> ::= "{" <instructions> "}"
<instrblock_exc> ::= "{" <instructions> <ehandlers>
                      <finally> "}"

<ehandlers> ::= <<empty>>
               | "exception" "(" <ident> ")" <instrblock>
               | "exception" "(" "..." ")" <instrblock>
```

```

<finally> ::= <<empty>>
           | "finally" <instrblock>

<dynobjs> ::= <dynobj> | <dynobj> <dynobjs>

<dynobj> ::= <funcsig> ";"
           | "exception" <ident> ";"
           | <record>

```

<signature> (*ld. 1.1.1. fejezet*)

<option> (*ld. 1.1.3. fejezet*)

<meta> (*ld. 1.1.4. fejezet*)

<block> (*ld. 1.2. fejezet*)

<instrblock> (*ld. 1.5.2. fejezet*)

<instrblock_exc> (*ld. 1.5.2. fejezet*) (*ld. 1.8. fejezet*)

<funcsig> (*ld. 1.3.1. fejezet*)

<instructions> (*ld. 1.5. fejezet*)

<ehandlers> (*ld. 1.8.1. fejezet*)

<finally> (*ld. 1.8.1. fejezet*)

<record> (*ld. 1.3.3. fejezet*)

A.2. Operátorok

A lusta operátorok kivételével az összes, előre definiált operátor túlterhelései megtalálható az alábbi leírásba (ld. 1.4.1.1. fejezet).

A.2.1. Infix operátorok

|| [Precedencia: 5, Asszociativitás: bal] Logikai vagy.

```
function "||"(bool, bool) : bool;
```

&& [Precedencia: 6, Asszociativitás: bal] Logikai és.

```
function "&&"(bool, bool) : bool;
```

| [Precedencia: 7, Asszociativitás: bal] Bitenkénti vagy.

```
function "|"(i32, i32) : i32;  
function "|"(u32, u32) : u32;  
function "|"(i64, i64) : i64;  
function "|"(u64, u64) : u64;
```

^ [Precedencia: 8, Asszociativitás: bal] Bitenkénti kizáró vagy.

```
function "^"(i32, i32) : i32;  
function "^"(u32, u32) : u32;  
function "^"(i64, i64) : i64;  
function "^"(u64, u64) : u64;
```

& [Precedencia: 9, Asszociativitás: bal] Bitenkénti és.

```
function "&"(i32, i32) : i32;  
function "&"(u32, u32) : u32;  
function "&"(i64, i64) : i64;  
function "&"(u64, u64) : u64;
```

== [Precedencia: 10, Asszociativitás: bal] Egyenlőségvizsgálat.

```
function "=="(bool, bool) : bool;  
function "=="(i32, i32) : bool;  
function "=="(u32, u32) : bool;  
function "=="(i64, i64) : bool;  
function "=="(u64, u64) : bool;  
function "=="(f32, f32) : bool;  
function "=="(f64, f64) : bool;  
function "=="(char, char) : bool;  
function "=="(handle, handle) : bool;  
function "=="(dhandle, dhandle) : bool;
```

!= [Precedencia: 10, Asszociativitás: bal] Nemegyenlőség-vizsgálat.

```
function "!="(bool, bool) : bool;  
function "!="(i32, i32) : bool;  
function "!="(u32, u32) : bool;  
function "!="(i64, i64) : bool;  
function "!="(u64, u64) : bool;  
function "!="(f32, f32) : bool;  
function "!="(f64, f64) : bool;  
function "!="(char, char) : bool;  
function "!="(handle, handle) : bool;  
function "!="(dhandle, dhandle) : bool;
```

< [Precedencia: 11, Asszociativitás: bal] Kisebb.

```
function "<"(i32, i32) : bool;  
function "<"(u32, u32) : bool;  
function "<"(i64, i64) : bool;  
function "<"(u64, u64) : bool;  
function "<"(f32, f32) : bool;  
function "<"(f64, f64) : bool;
```

> [Precedencia: 11, Asszociativitás: bal] Nagyobb.

```
function ">"(i32, i32) : bool;  
function ">"(u32, u32) : bool;  
function ">"(i64, i64) : bool;  
function ">"(u64, u64) : bool;  
function ">"(f32, f32) : bool;  
function ">"(f64, f64) : bool;
```

<= [Precedencia: 11, Asszociativitás: bal] Kisebb vagy egyenlő.

```
function "<=" (i32, i32) : bool;  
function "<=" (u32, u32) : bool;  
function "<=" (i64, i64) : bool;  
function "<=" (u64, u64) : bool;  
function "<=" (f32, f32) : bool;  
function "<=" (f64, f64) : bool;
```

>= [Precedencia: 11, Asszociativitás: bal] Nagyobb vagy egyenlő.

```
function ">=" (i32, i32) : bool;  
function ">=" (u32, u32) : bool;  
function ">=" (i64, i64) : bool;  
function ">=" (u64, u64) : bool;  
function ">=" (f32, f32) : bool;  
function ">=" (f64, f64) : bool;
```

« [Precedencia: 12, Asszociativitás: bal] Balra „shift”-elés.

```
function "<<"(i32, i32) : i32;  
function "<<"(u32, u32) : u32;  
function "<<"(i64, i64) : i64;  
function "<<"(u64, u64) : u64;
```

» [Precedencia: 12, Asszociativitás: bal] Jobbra „shift”-elés.

```
function ">"(i32, i32) : i32;  
function ">"(u32, u32) : u32;  
function ">"(i64, i64) : i64;  
function ">"(u64, u64) : u64;
```

+ [Precedencia: 13, Asszociativitás: bal] Összeadás.

```
function "+"(i32, i32) : i32;  
function "+"(u32, u32) : u32;  
function "+"(i64, i64) : i64;  
function "+"(u64, u64) : u64;  
function "+"(f32, f32) : f32;  
function "+"(f64, f64) : f64;  
function "+"(char, i32) : char;  
function "+"(string, string) : string;
```

- [Precedencia: 13, Asszociativitás: bal] Kivonás.

```
function "-"(i32, i32) : i32;  
function "-"(u32, u32) : u32;  
function "-"(i64, i64) : i64;  
function "-"(u64, u64) : u64;  
function "-"(f32, f32) : f32;  
function "-"(f64, f64) : f64;  
function "-"(char, char) : i32;
```

* [Precedencia: 14, Asszociativitás: bal] Szorzás.

```
function "*" (i32, i32) : i32;  
function "*" (u32, u32) : u32;  
function "*" (i64, i64) : i64;  
function "*" (u64, u64) : u64;
```

```
function "*" (f32, f32) : f32;  
function "*" (f64, f64) : f64;
```

/ [Precedencia: 14, Asszociativitás: bal] Osztás.

```
function "/" (i32, i32) : i32;  
function "/" (u32, u32) : u32;  
function "/" (i64, i64) : i64;  
function "/" (u64, u64) : u64;  
function "/" (f32, f32) : f32;  
function "/" (f64, f64) : f64;
```

% [Precedencia: 14, Asszociativitás: bal] Maradékképzés.

```
function "%" (i32, i32) : i32;  
function "%" (u32, u32) : u32;  
function "%" (i64, i64) : i64;  
function "%" (u64, u64) : u64;  
function "%" (f32, f32) : f32;  
function "%" (f64, f64) : f64;
```

A.2.2. Prefix operátorok

! [Precedencia: 15, Asszociativitás: jobb] Logikai negálás.

```
function "!" (bool) : bool;
```

~ [Precedencia: 15, Asszociativitás: jobb] Bitenkénti negálás.

```
function "~" (i32) : i32;  
function "~" (u32) : u32;  
function "~" (i64) : i64;  
function "~" (u64) : u64;
```

- [Precedencia: 0, Asszociativitás: jobb] Ellentett.

```
function "-"(i32) : i32;  
function "-"(u32) : u32;  
function "-"(i64) : i64;  
function "-"(u64) : u64;  
function "-"(f32) : f32;  
function "-"(f64) : f64;
```

- + [Precedencia: 0, Asszociativitás: jobb] Önmaga. *A háttérben nem fog történni semmi, ez csak egy szintaktikai kiegészítés, hogy szimmetrikus legyen az „ellentett”-tel.*

A.3. Névvel ellátott blokk használata

Ez a példaprogram bemutatja, hogyan lehet használni a névvel ellátott blokkokat.

```
#version 2.0

function func(i32 x) : i32
{
    f64 y;

    bo // ez a blokk neve
    {
        i32 y = x;

        while(y<0)
        bw // a ciklus blokk neve
        {
            string y = read();
            bo:y = bo:y + toi32(bw:y);
            // itt a 'bw:' el is hagyható
        }

        return y;
    }
}

main
b0
{
    i32 a = 0;
    i32 b = 0;
```

```

b1
{
    u32 a = tou32(b0:a);

    b2
    {
        string a = tos(b1:a);

        {
            b2:b = func(b0:a);
        }
    }
}

write(tos(b) + "\n");
}

```

Kimenet:

0

A.4. Elágazás példa **elif** nélkül

Ennek a példaprogramnak a célja, hogy bemutassa, hogyan történne egy sokágú elágazás akkor, ha nem létezne az **elif** (ld. 1.5.4. fejezet).

```
#version 2.0
main
{
    bool b1, b2, b3 = false, false, false;

    if(b1)
    {
        // 1.
    }
    else
    {
        if(b2)
        {
            // 2.
        }
        else
        {
            if(b3)
            {
                // 3.
            }
            else
            {
                // n.
            }
        }
    }
}
```

A.5. A foreach használata

Ez a program bemutatja, hogyan lehet használni a **foreach** ciklust egy sorozat bejárására. A sorozatot a **fill()** függvény tölt fel. A **fill()** függvény fordítva járja be az intervallumot, és a sorozat végéhez fűzi hozzá az új elemet.

A **main** blokkban található **foreach** 0-tól indul, és a ciklusváltozó-
nak kisebbnek kell lennie a sorozat hosszánál. Ezért van a ciklus felső
határa kizárva az intervallumból (**exclusive** módosító).

```
#version 2.0

function fill(u32 n) : seq<u32>
{
    seq<u32> s;

    foreach(i in 1u..n reverse)
    {
        s = hiext(s, i);
    }

    return s;
}

main
{
    seq<u32> sorozat = fill(12u);

    foreach(i in 0u .. seqlen(sorozat) exclusive)
    {
        write(tos(sorozat[i]) + "\n");
    }
}
```

Kimenet:

12
11
10
9
8
7
6
5
4
3
2
1

A.6. Példa operátor túlterhelésekre

Az alábbi program túlterheli a `<` operátort, ami össze tud hasonlítani egy előjeles és egy előjel nélküli 32 bites egész számot. Ha a baloldali negatív, vagy a jobboldali nagyobb, mint a legnagyobb előjeles pozitív 32 bites szám, akkor az összehasonlítás eredménye igaz. A teljesség kedvéért a lehetséges kivételt lekezezi a függvény, kiír egy üzenetet a szabványos kimenetre, és az összehasonlítás eredménye hamis lesz. A kivételkezelő sose fog lefutni, mert az összes problémás esetet lekezelte az elágazásban (a konverzió akkor váltana ki kivételt, ha a száma kisebb lenne nullánál).

A kód tartalmaz további két operátor túlterhelést. Mindkettő a `+` operátor túlterhelése. Az első 4 db előjel nélküli 32 bites egész számot vár paraméterül, és az eredmény 2 db ugyanilyen szám lesz. A másik túlterhelés egy **string**-es és egy **u32** típusú értéket vár paraméterül. Ezt arra használja a kód, hogy a két **u32** típusú kifejezések szöveggé alakítását minél egyszerűbben lehessen leírni.

A **main** blokkban ki is próbálja az új operátorokat. Az érdekesebb eset a 4 db **u32** paramétert váró operátor. Két módon történik az operátor hívása, de mind a kettő ugyanazt jelenti. Az első mód a tényleges hívás, vagyis a függvényhívás. A második mód operátorral hívja meg ugyanazt a függvényt – a zárójelek azért kellenek, hogy a jobb- és a baloldalon is **u32**, **u32** típusú kifejezés álljon. Végül megjeleníti a kiszámított értéket a szabványos kimeneten.

```
#version 2.0

import
{
    // A kivételt importálni kell!
    exception ConstraintErrorException;
}
```

```

function "<"(i32 a, u32 b) : bool
{
    if(a<0 ||| b>0xffffffffu) { return true; }
    return tou32(a) < b;

    exception(ConstraintErrorException)
    {
        write("ConstraintErrorException\n");
        return false;
    }
}

function "+"(u32 a, u32 b, u32 c, u32 d) : u32, u32
{
    return a+c, b+d;
}

function "+"(string s, u32 x) : string
{
    return s + tos(x);
}

function tos(u32 a, u32 b) : string
{
    return "(" + a + ", " + b + ")";
    // Felhasznált operátorok (balról jobbra):
    //   "+"(string, u32)
    //   "+"(string, string)
    //   "+"(string, u32)
    //   "+"(string, string)
}

```

```
main
{
    write(tos(-10<10u) + "\n");

    u32 a, b = "+"(1u, 2u, 3u, 4u);
    u32 A, B = (1u, 2u) + (3u, 4u);
    // a==A && b==B
    write(tos(a, b) + "\n");
}
```

Kimenet:

```
true
(4, 6)
```

A.7. Lokális változók létrehozása

A következő példa bemutatja, hogyan lehet lokális változókat deklarálni. A kezdő értékadás nélküli változókat a típus kezdőértékére (ld. 2.1. fejezet) állítja. Ha van kezdő értékadás, akkor kötelező az összes értéket inicializálni.

```
#version 2.0

global
{
    string gs = "glob";
}

main
{
    i32 x, y, z; // kezdőérték: 0
    write(tos(x) + ", " + tos(y) + ", " + tos(z) + "\n");

    u32 X, Y, Z = 3u, 4u, 5u;
    write(tos(X) + ", " + tos(Y) + ", " + tos(Z) + "\n");

    string a, b, c = "aa" + "aa", gs, global:gs;
    write(a + ", " + b + ", " + c + "\n");

    string A; // kezdőérték: ""
    A = a;
    write(A + "\n");
}
```

Kimenet:

```
0, 0, 0
```

```
3, 4, 5
```

```
aaaa, glob, glob
```

```
aaaa
```

A.8. Szabványos kivételek

Ez a példaprogram a szabványok kivétel lekezelésével foglalkozik (*ld. 1.8.3. fejezet*). A leggyakrabban előforduló kivételek keletkezési helyét demonstrálja kisarkított példákkal. Az aktuális fordítóprogram nem tud olyan kódot generálni, amely az összes szabványos kivétel ki tudja váltani, mert még nincs implementálva az összes funkció, ami a kivételeket használná. Továbbá a ***UnknownException*** kivétel csak egy helyfoglaló, ténylegesen sehol sincs használva.

```
#version 2.0

function u() : bool
{
    // nincs 'return'
    // UnhandledControlPathException
}

function U()
{
    bool b = u();

    exception(UnhandledControlPathException)
    {
        write("UnhandledControlPathException\n");
    }
}
```

```

function C()
{
    u32 x = tou32(-1); // ConstraintErrorException

    exception(ConstraintErrorException)
    {
        write("ConstraintErrorException\n");
    }
}

function Z()
{
    i32 z = 1.f; z = z - z;
    z = 1.f / z; // DivisionByZeroException

    exception(DivisionByZeroException)
    {
        write("DivisionByZeroException\n");
    }
}

function I()
{
    string s = "abc";
    char ch = s[10u]; // IndexOutOfRangeException

    exception(IndexOutOfRangeException)
    {
        write("IndexOutOfRangeException\n");
    }
}

```

```
main
```

```
{
```

```
    U();
```

```
    C();
```

```
    Z();
```

```
    I();
```

```
}
```

Kimenet:

```
UnhandledControlPathException
```

```
ConstraintErrorException
```

```
DivisionByZeroException
```

```
IndexOutOfRangeException
```

A.9. Lusta „és” és lusta „vagy”

A következő a lusta operátorok használatát mutatja be.

Az [1]-es rész a lusta „és” operátort használja. Ha nem lenne lusta az operátor, akkor a **string** indexelése ***IndexOutOfRangeException*** kivételt váltana ki, és terminálna a program.

A [2]-es rész a lusta „vagy” operátort használja. A **FastIsPrime** függvény először teszteli, hogy a kapott szám egyjegyű prím-e, és ha igen, akkor **true** lesz a visszatérési érték. A kifejezés egy lusta „és”-sel végződik, ami csak akkor hívja meg a **IsPrime** függvényt, ha a szám nem páros (2 nem lehet, mert azt már a lusta „vagy” kiszűrte).

```
#version 2.0

function fourthD(string s) : bool
{
    return strlen(s)>3 &&& s[3u]=='D';
}

function IsPrime(i32 n) : bool
{
    if(n<2) { return false; }

    i32 i;
    i32 r = sqrt(n); // [3]

    for(i=2;i<=r;i=i+1)
    {
        if((n%i)==0) { return false; }
    }

    return true;
}
```

```

function FastIsPrime(i32 n) : bool
{
    return (n==2) ||| (n==3) ||| (n==5) ||| (n==7)
        ||| ((n%2)!=0 &&& IsPrime(n));
}

main
{
    // [1]
    write("---- 1 ----\n");
    write(tos(fourthD("")) + "\n");
    write(tos(fourthD("abc")) + "\n");
    write(tos(fourthD("abcd")) + "\n");
    write(tos(fourthD("abcD")) + "\n");

    // [2]
    write("---- 2 ----\n");
    write(tos(FastIsPrime(1)) + "\n");
    write(tos(FastIsPrime(2)) + "\n");
    write(tos(FastIsPrime(4)) + "\n");
    write(tos(FastIsPrime(7)) + "\n");
    write(tos(FastIsPrime(880000000)) + "\n");
}

```

Kimenet:

```
---- 1 ----  
false  
false  
false  
true  
---- 2 ----  
false  
true  
false  
true  
false
```

A.10. Gyorsrendezés

Az alábbi kódhoz tartozó megjegyzések az eredmények fejezetben találhatók (*ld. 4.3.2. fejezet*).

```
#version 2.0

declare
{
    function qsort(mutable seq<i32> arr, u32 beg, u32 end);
}

function qsort(mutable seq<i32> arr)
{
    u32 len = seqlen(arr);
    if(len>1u)
    {
        qsort(arr, 0u, len);
    }
}

function qsort(mutable seq<i32> arr, u32 beg, u32 end)
{
    if(end>beg+1u)
    {
        i32 piv = arr[beg];
        u32 l, r = beg+1u, end;
```

```

        while(l<r)
        {
            if(arr[l]<=piv)
            {
                l = l + 1u;
            }
            else
            {
                r = r - 1u;
                arr[l], arr[r] = arr[r], arr[l];
            }
        }

        l = l - 1u;
        arr[l], arr[beg] = arr[beg], arr[l];

        qsort(arr, beg, l);
        qsort(arr, r, end);
    }
}

main
{
    seq<i32> arr = seqfill(0, 10u);

    arr[0u] = 8; arr[1u] = 4;
    arr[2u] = 3; arr[3u] = 0;
    arr[4u] = 9; arr[5u] = 6;
    arr[6u] = 2; arr[7u] = 5;
    arr[8u] = 1; arr[9u] = 7;

    qsort(arr);
}

```

```

foreach(i in 0u .. seqlen(arr) exclusive)
{
    if(i>0u) { write(", "); }
    write(tos(arr[i]));

}

write("\n");
}

```

Kimenet:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

A.10.1. C implementáció

```

void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void sort(int arr[], int beg, int end)
{
    if(end>beg+1)
    {
        int piv = arr[beg], l = beg+1, r = end;

```

```
while(l<r)
{
    if(arr[l]<=piv)
    {
        ++l;
    }
    else
    {
        swap(&arr[l], &arr[--r]);
    }
}

swap(&arr[--l], &arr[beg]);

sort(arr, beg, l);
sort(arr, r, end);
}
}
```

A.11. Külső C-s könyvtárak felhasználása

Egy rövid példán keresztül bemutatásra kerül a C-s programkönyvtárak felhasználásának, beépítésének menete. A példában használt két függvény:

- **int chdir(const char* path);**
- **double pow(double x, double y);**

A kódban a „szokásos” típusokat használom, mert az **int** ekvivalens az **i32**-vel, és a **double** az **f64** típussal.

```
#version 2.0

import
{
    function c_chdir(string path) : int;
    function c_pow(double x, double y) : double;
}

main
{
    int r = c_chdir("..");
    write("c_chdir(\"..\"): " + tos(r) + "\n");

    double p = c_pow(2.0, 10.0);
    write("c_pow(2.0, 10.0): " + tos(p) + "\n");
}
```

Példa kimenet:

```
c_chdir(".."): 0
c_pow(2.0, 10.0): 1024.000000
```

A.11.1. Függvény wrapper-ek

A használatához el kell készíteni C nyelvben a wrapper függvényeket. A wrapper függvények törzse a virtuális gép API-ját kell használni, hogy a paramétereket ki lehessen szedni az adatveremből, és az eredményeket vissza lehessen rakni az adatverembe.

Fontos, hogy a kapott paramétereket fordított sorrendben kell ki-szedni a veremből.

```
void chdir_wrapper(sk_program program)
{
    sk_cstring path = skProgram_Pop_string(program);
    int r = chdir(path);
    skProgram_Push_i32(program, r);
}

void pow_wrapper(sk_program program)
{
    sk_f64 y = skProgram_Pop_f64(program);
    sk_f64 x = skProgram_Pop_f64(program);
    sk_f64 r = pow(x, y);
    skProgram_Push_f64(program, r);
}
```

Ezeket a függvényeket be kell illeszteni az aktuális környezetbe.

```
{
    sk_u32 a_s[] = { SKVT_STRING };
    sk_u32 a_i[] = { SKVT_I32 };
    sk_u32 a_d[] = { SKVT_F64 };
    sk_u32 a_dd[] = { SKVT_F64, SKVT_F64 };

    skProfile_InsertFunction(profile, "c_chdir",
        1, a_s, 1, a_i, chdir_wrapper);

    skProfile_InsertFunction(profile, "c_pow",
        2, a_dd, 1, a_d, pow_wrapper);
}
```

A.12. A virtuális gép utasításkészlete

A virtuális gép utasításainak rövid leírása. A leírásban használt **opdata** az utasítás 24 bites „*immediate*” adatát jelenti (az utasításba kódolt 24 bites adat).

Egy utasítás 32 bitből áll. Az utasítás (mint egész szám) 8 legmagasabb helyiértékű bitje az „*opcode*”, vagyis az utasítás kódja. Az alsó 24 bit pedig az utasítás paramétere. Ezt a paramétert az assembler állítja be, és a magas szintű nyelvből lefordított program ezt a paramétert nem változtatja meg¹.

A leírásban használt **PC** a „*program counter*”-t jelenti, ami mindig a következő utasítás címét tárolja (a **PC** értéke utasításokban értendő).

A.12.1. Speciális és általános utasítások

nop „*No operation.*” Nincs művelet. Helykitöltéshez használt.

stop Végrehajtás megállítása. Ezt az utasítást a virtuális gép használja belső mikroprogramokban. Az assembler nem támogatja ezt az utasítást.

idrd „*Internal DeReference in Datastack.*” Adatvermen belüli dereferálás. Kiolvassuk az adatverem tetejétől **opdata** távolságban levő címről egy 32 bites értéket, és erre lecseréljük a címet, amiről kiolvastuk.

A.12.2. Verem vezérlő utasítások

pusham Elhelyezi a verem tetején az **opdata+AM** 32 bites értéket. Az **AM** a program adatterületének abszolút címe.

pushdvt Elhelyezi a verem tetején az **opdata+DVT** 32 bites értéket. A **DVT** az aktuális változó táblázat (*ld. 3.1.2. fejezet*).

¹Az utasítások is a memóriában helyezkednek el, és lehet olyan kódot írni assemblyben, amivel az utasítások paraméterei módosíthatók.

store Kiolvas az adatveremről egy 32 bites címet, és arra a címre lementi az adatverem felső **opdata** byte-ját. A művelet után törli az elmentett adatot.

load Kiolvas az adatveremről egy 32 bites címet, és arról a címről betölt és elhelyez az adatveremre **opdata** byte-ot.

duplicate Duplikálja az adatverem felső **opdata** byte-ját.

pop Törli az adatverem felső **opdata** byte-ját.

push32 Elhelyezi az **opdata**-t, mint 32 bites egész számot az adatvermen.

push32neg Elhelyezi az **opdata**-t, mint 1-esekkel kiegészített 32 bites egész számot az adatvermen.

push64 u.a. mint a **push32**, de 64 bites értékkel.

push64neg u.a. mint a **push32neg**, de 64 bites értékkel.

dvtalloc Létrehozza az **opdata** byte méretű **DVT**-t.

dvtfree Törli a **DVT**-t.

csosalloc Létrehozza az **opdata** elem kapacitású **CSOS**-t.

csospush Berak a **CSOS**-be egy bejegyzést az adatveremről.

csospop Kivesz egy bejegyzést a **CSOS**-ból, és az adatvermen tárolja el.

csossp Elhelyezi a **CSOS** elemszámát az adatvermen, mint 32 bites szám.

csosfree Törli a **CSOS**-t.

tossiz Elhelyezi a **TOS** elemszámát az adatvermen, mint 32 bites szám.

tospop Kivesz egy bejegyzést a **TOS**-ból, és az adatvermen tárolja el.

tospush Berak a **TOS**-be egy bejegyzést az adatveremről.

tosclear Kitörli az aktuális jelzésig a **TOS**-t.

tosdrop Eldobja a **TOS** legfelső bejegyzését.

pushdsp Berakja az adatverembe a **DSP-opdata** címet.

A.12.3. Ugró utasítások

jmp Feltétel nélküli ugrás. A **PC**-hez hozzáadja az **opdata**-t.

jmp1 Feltétel nélküli ugrás. Kivesz az adatveremből egy 32 bites értéket, és ezt az értéket hozzáadja a **PC**-hez.

jmp2 Kivesz az adatveremből egy 32 bites értéket, és ha az érték nem 0, akkor hozzáadja a **PC**-hez az **opdata**-t.

jmp3 Kivesz az adatveremből két 32 bites értéket, és ha az első érték nem 0, akkor hozzáadja a **PC**-hez a második értéket.

jmp4 Kivesz az adatveremből egy 32 bites értéket, és ha az érték 0, akkor hozzáadja a **PC**-hez az **opdata**-t.

jmp5 Kivesz az adatveremből két 32 bites értéket, és ha az első érték 0, akkor hozzáadja a **PC**-hez a második értéket.

jmp6 Felülírja a **PC**-t az **opdata**-val.

jmp7 Kivesz az adatveremből egy címet, és felülírja vele a **PC**-t.

A.12.4. Hívással kapcsolatos utasítások

enter Meghívja az **opdata** lokális című függvényt.

leave Kilép az aktuális függvényből.

dle opdata sorszámú dinamikusan linkelt függvény végrehajtása.

exec Végrehajtja az **opdata** sorszámú beépített függvényt.

mkfpenter Létrehoz egy **enter** alapú függvény mutatót.

mkfpdle Létrehoz egy **dle** alapú függvény mutatót.

callfptr Meghívja az **mkfpenter** vagy **mkfpdle** utasításokkal létrehozott függvény mutatót.

pushpc Berakja az adatverembe az aktuális **PC**-t.

A.12.5. Objektum vezérlés

strinit Inicializálja az adatveremről kivett című **string**-et az **AM+opdata** című karaktersorozattal.

strdelete Törli az adatveremről kivett című **string**-et.

strnew Az adatverem tetején lévő címen található karaktersorozatról készít egy másolat **string**-et, és a másolat címével felülírja az adatverem felső 4 byte-ját.

strlen Meghatározza az adatverem tetején lévő **string** hosszát, és felülírja vele az adatverem felső 4 byte-ját.

seqnew Létrehoz egy üres **opdata** méretű elemeket tartalmazó sorozat vázat.

seqdelete Törli a megadott sorozat vázat.

seqlength Meghatározza a sorozat váz hosszát.

seqitmrefi Meghatározza az **opdata** indexű elem címét.

seqitmrefl Meghatározza az adatveremből kivett indexű elem címét.

recnew Létrehoz egy **opdata** méretű rekordot.

recdelete Törli a megadott rekordot.

recfldref Meghatározza az **opdata** ofszetű mező címét.

A.12.6. Párhuzamosság

dqi Ha az **opdata** 0, akkor csökkenti, egyébként növeli a *Disable Quantum Interrupt* szintet. **Ezt az utasítást nem használja a fordító.**

A.12.7. Kivételek

epush opdata lokális indexű kivétel elhelyezése az adatvermen.

eraise Adatverem tetején található (globális) kivétel kiváltása.

ehandler opdata lokális című utasítás beállítása kivételkezelőnek. A beállítás csak egy *call frame* (ld. 3.1.2. fejezet).

ehclear Az aktuális kivételkezelő beállítás törlése.

A.12.8. Aritmetikai utasítások

add Két 32 bites szám összeadása.

addi Egy 32 bites számhoz hozzáadja az **opdata** értékét.

sub Két 32 bites szám kivonása.

subi Egy 32 bites számból kivonja az **opdata** értékét.

shl Logikai shift-elés balra.

shli Logikai shift-elés balra **opdata**-val.

shr Logikai shift-elés jobbra.

shri Logikai shift-elés jobbra **opdata**-val.

lsor Balra shift-el egy 32 bites számot 16 bittel, és bitenként hozzáavgolja az **opdata**-t (ami csak 16 bites lehet).

and Bitenkénti és.

or Bitenkénti vagy.

xor Bitenkénti kizáró vagy.

neg Bitenkénti negálás.

not Logikai negálás.

rsbi Az **opdata** értékéből von ki egy 32 bites számot.

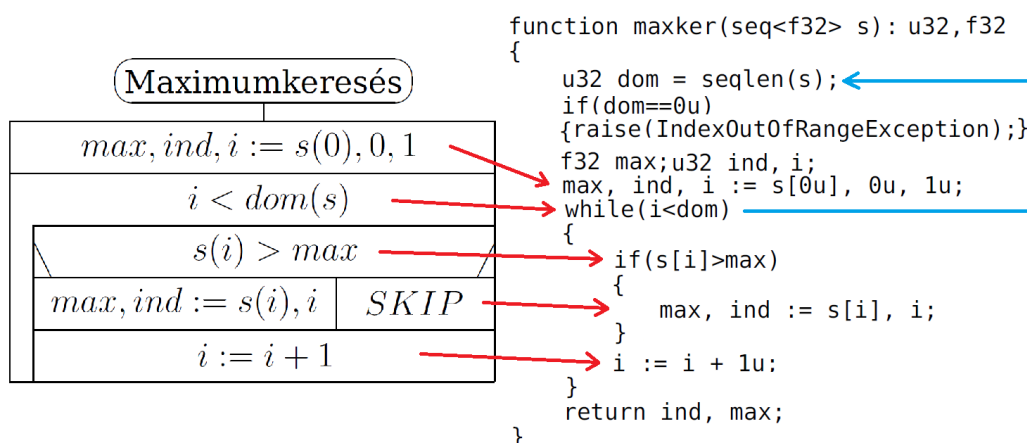
A.13. Példa algoritmus implementálásra

A dolgozatban bemutatott maximumkeresés algoritmus implementálását (*ld. 4.2.1. fejezet*) illusztrálja az A.1 ábra.

Az ábrán látható piros nyilak azt mutatják, hogy az implementáció lényegében csak az absztrakt algoritmus egyes lépéseinek átmásolása a forráskódba. Megfigyelhető, hogy a `:=` jelek is megegyeznek, mert a szintaxis megengedi a `=` és `:=` jelek használatát is. A kék nyíl egy triviális optimalizáció: a ciklusfeltétel hivatkozik a sorozat hosszára, de ez az érték nem fog megváltozni a ciklusmagban, ezért ezt ki lehet emelni egy lokális változóba.

Néhány formai követelményt azonban nem elégít ki az absztrakt algoritmusban használt jelölésrendszer: a sorozat egyes elemeihez nem gömbölyű zárójellel kell hozzáférni, hanem szögletes zárójellel, valamint a konstansok után ki kell írni a típusjelzést.

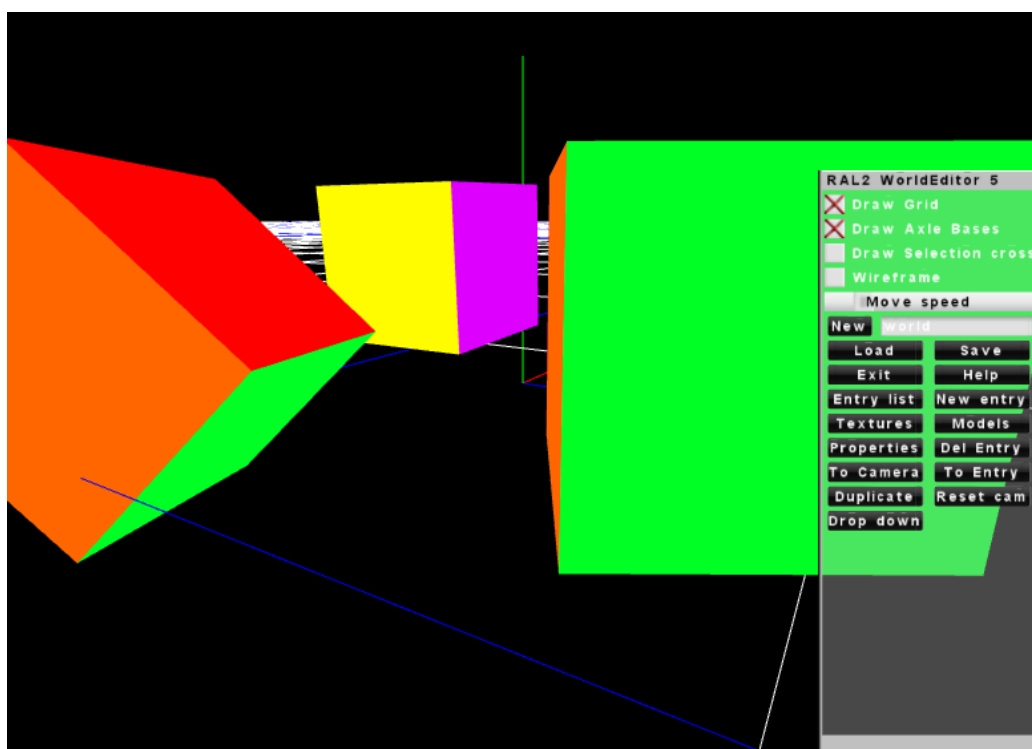
A függvény elején van egy ellenőrzés, hogy a sorozat nem lehet üres. Ezt a feltételt át lehetne tenni egy előfeltételvizsgálatba is (*ld. 5.2.1. fejezet*).



A.1. ábra. A maximumkeresés algoritmus implementálása.

A.14. Képernyőkép a pályaszerkesztőből

A saját nyelvvel készített eddigi legnagyobb programom, a 3-dimenziós pályaszerkesztő (ld. 4.3.3. fejezet) egy pillanatképe látható az A.2 ábrán.



A.2. ábra. Pillanatkép a pályaszerkesztőről.