

# ScriptKernel – beágyazott domain specifikus nyelvek megvalósítása

---

Készítette: Baráth Áron  
Programtervező Informatikus, MSc.  
ELTE

Témavezetők: Dr. Kozsik Tamás, Szűgyi Zsolt  
ELTE, Informatikai Kar,  
Programozási nyelvek és Fordítóprogramok Tanszék



---

2013

## **Absztrakt**

A mai világban egyre elterjedtebbek a domain specifikus nyelvek, mert számos előnyük van az általános célú programozási nyelvekkel szemben. A legfontosabb tulajdonságuk az, hogy a programot a probléma szintjén lehet megfogalmazni, és ugyanígy verifikálni is. Azonban a domain specifikus nyelveknek árnyoldala is van, amely a magas kifejlesztési és betanulási költségekben mutatkozik meg. Erre egyfajta megoldás adnak a beágyazott domain specifikus nyelvek, mert így egy gazdanyelvre támaszkodva kevésbé költséges azok kifejlesztése, valamint az így elkészült domain specifikus nyelvek valamennyire hasonlítanak egymásra, ami a betanulás költségeit is csökkenti. Fontos szempont egy megfelelő gazdanyelv kiválasztása – fontos az erős statikus típusrendszer és a platform függetlenség. Ilyen kritériumokkal rendelkezik a saját fejlesztésű programozási nyelvem, amely kitűnő gazdanyelv beágyazott domain specifikus nyelvek fejlesztéséhez. Ez a dolgozat a domain specifikus nyelvek implementációjának egy új aspektusát mutatja be, felhasználva a nyelv erős statikus típusrendszerét és az operátor mechanizmust.

# Tartalomjegyzék

|                                                                        |           |
|------------------------------------------------------------------------|-----------|
| <b>Bevezető</b>                                                        | <b>5</b>  |
| <b>1. Áttekintés a ScriptKernel nyelvről</b>                           | <b>10</b> |
| 1.1. Nyelvi elemek . . . . .                                           | 11        |
| 1.2. Típusok . . . . .                                                 | 13        |
| <b>2. A domain specifikus nyelvek megvalósítását támogató eszközök</b> | <b>15</b> |
| 2.1. Privát rekordok . . . . .                                         | 15        |
| 2.2. Operátorok . . . . .                                              | 17        |
| 2.3. Program frissítése futás közben . . . . .                         | 21        |
| <b>3. Domain specifikus nyelv megvalósítás ScriptKernel nyelven</b>    | <b>24</b> |
| 3.1. Komplex számok megvalósítása . . . . .                            | 24        |
| 3.2. OpenGL . . . . .                                                  | 28        |
| 3.3. Petri-háló . . . . .                                              | 33        |
| 3.3.1. Háló létrehozása . . . . .                                      | 35        |
| 3.3.2. Háló szimulálása . . . . .                                      | 37        |
| 3.3.3. Csokiautomata . . . . .                                         | 38        |
| <b>Összegzés</b>                                                       | <b>41</b> |
| Továbbfejlesztési lehetőségek . . . . .                                | 42        |
| <b>Irodalomjegyzék</b>                                                 | <b>44</b> |

|                                                                                               |           |
|-----------------------------------------------------------------------------------------------|-----------|
| <b>Függelék A.</b>                                                                            | <b>47</b> |
| A.1. Nyelvi elemek . . . . .                                                                  | 47        |
| A.1.1. Blokkok . . . . .                                                                      | 47        |
| A.1.2. Deklarációk . . . . .                                                                  | 48        |
| A.1.3. Kifejezések . . . . .                                                                  | 53        |
| A.1.4. Értékadás . . . . .                                                                    | 53        |
| A.1.5. Kivételkezelés és -kiváltás . . . . .                                                  | 55        |
| A.1.6. Elő- és utófeltételek . . . . .                                                        | 56        |
| A.1.7. Elágazás . . . . .                                                                     | 58        |
| A.1.8. Ciklusok . . . . .                                                                     | 59        |
| A.2. Típusok . . . . .                                                                        | 62        |
| A.2.1. Alaptípusok . . . . .                                                                  | 62        |
| A.2.2. Rekordok . . . . .                                                                     | 64        |
| A.2.3. Sorozatok . . . . .                                                                    | 65        |
| A.2.4. A <b>mutable</b> . . . . .                                                             | 66        |
| A.3. <b>complex.sks</b> : Komplex számok típusa . . . . .                                     | 68        |
| A.3.1. A fordító által generált import include . . . . .                                      | 69        |
| A.4. <b>128.sks</b> : A 128 bites előjel nélküli egész típus . . . . .                        | 70        |
| A.4.1. A fordító által generált import include . . . . .                                      | 76        |
| A.5. <b>test.sks</b> : A 128 bites egész típus tesztprogramja . . . . .                       | 78        |
| A.6. <b>gl.sks</b> : OpenGL példaprogram . . . . .                                            | 78        |
| A.6.1. A fordító által generált import include . . . . .                                      | 82        |
| A.7. <b>gltest.sks</b> : Az OpenGL példaprogram használata . . . . .                          | 83        |
| A.8. <b>petri_demo.sks</b> : A Petri-háló DSL használata . . . . .                            | 85        |
| A.9. <b>csoki.sks</b> : Csokiautomata implementálása Petri-háló<br>DSL segítségével . . . . . | 86        |

# Bevezető

A domain specifikus nyelveket (*DSL*) egyre elterjedtebb körökben alkalmazzák, köszönhetően annak, hogy az ilyen módon megírt programok a probléma szintjén validálhatók. Ez a programok minőségét is javítja. A DSL nyelvek egyik fő hátránya, hogy a kifejlesztésükre és betanulásukra fordított erőforrások túl magasak a szűk felhasználási területhez képest. Ezen a problémán segít az, ha a DSL nyelveket nem a lexikális elemzőtől kell minden egyes alkalommal felépíteni, hanem egy meglévő nyelvet használnak fel erre a célra. Az ilyen módon elkészített DSL nyelvet beágyazott DSL nyelvnek hívják. Előnye az, hogy az így módon elkészített DSL-ek valamennyire hasonlítanak egymásra, így a betanulásuk nem a lexikális elemektől kezdődik, hanem a DSL lényegi résszel: a használható típusokkal és a hozzájuk tartozó műveletekkel.

A domain specifikus nyelvek (*DSL*) speciális célra kifejlesztett nyelvek. Ezen nyelvek segítségével a probléma absztrakciós szintjén lehet dolgozni, ami minden DSL-t teljesen egyedivé teszi. Így minden speciális feladat megoldásához más és más DSL-t kell a felhasználóknak megtanulniuk. Gondot okozhat az is, hogy ugyanazt a feladatot többféleképpen is meg lehet közelíteni, így előfordulhat, hogy két, ugyanahhoz a feladathoz készült DSL nyelv teljesen különbözik. Ebből adódik az a probléma, hogy egyik DSL-ről nehéz az átállás egy másik, ugyanazt a problémát megoldó DSL-re.

A beágyazott domain specifikus nyelvek (*eDSL*) már meglévő programozási nyelveket használnak fel, egészítenek ki a domain specifikus eszköztár kialakítására. Ezek az *eDSL*-ek felhasználják a gazdanyelv infrastruktúráját, így nem kell minden *eDSL* készítésekor a lexikális elem-

zővel kezdeni a fejlesztést. Ez egyrészt a fejlesztési időt csökkenti, valamint az elkövethető hibákat is redukálja – ezáltal a gazdanyelv komponenseinek egyfajta kód újrafelhasználása történik.

A beágyazott DSL-ek előnye, hogy az ugyanarra a gazdanyelvre készített DSL-ek hasonlítanak egymásra, mert a DSL fejlesztője a gazdanyelv ugyanazon technikáit alkalmazza. Ezáltal a DSL-ek fejlesztése és betanulása sem a kezdetektől indul, ami több szempontból is fontos. Egyrészt csökkenti a fejlesztési és tanulási költségeket, másrészt fel lehet használni a már meglévő eszközöket. Hátrány viszont az, hogy a gazdanyelv stílusához kötődni fog a DSL is, ami egyfajta szintaktikai rugalmatlanságot von maga után.

Jelenleg a legelterjedtebb gazdanyelv a beágyazott domain specifikus nyelvek készítéséhez a Haskell [Hut07, OSG08] illetve a Scala [OSV08] programozási nyelv. Ezek a funkcionális nyelvek helyes és hatékony használata nagy rutint igényel, ezért nem terjedtek el – azonban az imperatív nyelvek jóval gyakoribb, valamint sokak számára érthetőbbek és átláthatóbbak. Továbbá a programozás folyamatát a nyelv statikus típusrendszere is segíti. Ilyen a saját fejlesztésű imperatív programozási nyelvem is, amely a ScriptKernel nevet viseli.

A DSL nyelvekben nagy előnyt jelent, ha a nyelv erősen típusos. A Haskell, és a saját nyelvem is erős statikus típusrendszerrel dolgozik. Ez azért fontos, mert a (beágyazott) DSL egyik alappillére a típusok, és a típusokon elvégezhető műveletek. Ezen műveletek megvalósítása a legtöbb esetben a függvények és operátorok túlterhelésével történik. Ehhez szükséges a jól kialakított statikus típusrendszer.

Dolgozatom célja annak bemutatása, hogyan használható beágyazott domain specifikus nyelvek implementációjához a saját fejlesztésű imperatív programozási nyelvem. Ez egy jó alternatíva olyanok számára, akik nem olyan rutinosak a funkcionális programozásban, vagy inkább az imperatív nyelveket részesítik előnyben. A nyelv legfontosabb tulajdonságai a szigorú szintaxis és szemantika, valamint ehhez kapcsolódóan az erős statikus típusrendszer. Az így kapott szigorú szabályok célja az, hogy a lehető legtöbb hibát még fordítási időben detektáljon

a fordító, és így a program megbízhatóbb lesz, valamint kevesebb futásidejű ellenőrzést követel meg, ami a program futási sebességét jótékonyan érinti. Azonban a kivételes események nem kerülhetők el, így a nyelv biztosítja a kivételek kiváltását és lekezelését egyaránt.

A nyelv röviden bemutatásra kerül az 1. fejezetben – áttekintést kapunk a forrás felépítéséről, a használható nyelvi elemekről és a nyelv típusrendszeréről. A beágyazott domain specifikus nyelvek fejlesztésével és felhasználásával kapcsolatos nyelvi elemek, konstrukciók és tulajdonságok a 2. fejezetben vannak részletezve.

### **Motiváció.**

A beágyazott domain specifikus nyelv feladata az esetleg felhasznált külső programkönyvtár helyes használata is. Vannak olyan programkönyvtárak, ahol az invariánsok dokumentációban vagy csak megjegyzések formájában állnak rendelkezésre. Ebben az esetben az eDSL-t úgy kell felépíteni, hogy ezeket az invariánsokat betartsa, tehát a programkönyvtár felhasználása helyes módon történjen. Ez a megfelelő típusokkal és az eDSL műveleteivel egyszerűen megvalósítható. A 3.2 fejezetben az OpenGL egy lehetséges megvalósítását ismertetem.

Az OpenGL alapvetően egy C nyelvű programkönyvtár (létezik más nyelvhez is, de az „eredeti” a C nyelvű). A kirajzolás során ügyelni kell arra, hogy a függvényhívások a megfelelő sorrendben történjenek. A legkritikusabb a primitív kirajzoláshoz használt függvénypár, a **glBegin()** és a **glEnd()**. A kirajzolandó primitív típusát a **glBegin()** paramétereként kell megadni, és a kirajzolás végén a paraméter nélküli **glEnd()** függvényt kell meghívni. Ezt a függvénypárt blokkszerűen kell használni, nem lehet egymásba ágyazni őket, valamint nem lehet átlapolni ezeket a „blokkokat”. Természetesen ez csak egyetlen példa a sok szabály közül, amelyeket be kell tartani ahhoz, hogy a megírt program jól működjön (az OpenGL egy nyílt szabvány, melynek implementációja gyártófüggő, ezért csak az a program fog jól viselkedni, amelyik az OpenGL szabványban leírtaknak megfelel – előfordulhatnak olyan esetek, amikor egy szabvány szerint hibás program nem műkö-

dik rosszul az egyik gyártó meghajtójával (*driver*), de egy másik gyártó meghajtójával már hibás).

Akik sokat dolgoztak OpenGL-lel, azok tudják, hogy készíteni kell egy magasabb szintű program könyvtárat a biztonságos használathoz. Már a 90-es évek végén John Carmack [iS97] se használta az OpenGL-t közvetlenül – csak célfüggvényekben voltak használva, kontrollált körülmények között. Például a `gl_draw.c` forrásban található kép kirajzoló függvény, ami az 1. ábrán olvasható.

Már a `Draw_Pic()` függvényben is több dologra kellett ügyelni: a belső állapottól függően ki kellett kapcsolni az alfa tesztet (ez a funkció csak akkor jelenített meg egy fragment-et, ha a fragment alfa értéke megfelelt egy feltételnek), valamint a kirajzolás után újra be kellett kapcsolni az eredeti állapot visszaállítása céljából. Én is foglalkoztam három dimenziós játékok programozásával, és sokkal kényelmesebbnek és biztonságosabbnak bizonyult a használat segédfüggvények bevezetésével. C++ nyelven operátorok segítségével lehet készíteni egy egyszerűbb szintaxist OpenGL-es kirajzolásokhoz. A probléma ezzel az, hogy nem lehet fordítási időben ellenőrizni azt, hogy a kirajzolást záró `draw::end` szerepel a kódban.

```
draw::triangle tri;  
// ...  
draw::begin() << tri << draw::end;
```

Elképzelhetnénk egy olyan megoldást, melyben a kirajzoló operátorok egy kirajzolási állapotot leíró objektumot adnának vissza és a `draw::end` pedig `void`-ot. Azonban ez a legtöbb nyelv, így a C++ esetén sem jó megoldás, mert egy kifejezés által visszaadott értéket nem kötelező felhasználni. Ebből kifolyólag nem lehet kikényszeríteni a `draw::end` használatát.

Ezért fontosak a DSL nyelvek, hogy a felmerülő laza, gazdanyelvi szinten nehezen ellenőrizhető szabályokat a megfelelő absztrakciós szintre emelve triviális módon felügyelje. Egy ilyen lehetőség a beágyazott DSL nyelvek implementációja a ScriptKernel nyelv segítségével.

```

void Draw_Pic (int x, int y, char *pic)
{
    image_t *gl;

    gl = Draw_FindPic (pic);
    if (!gl)
    {
        ri.Con_Printf (PRINT_ALL, "Can't find pic: %s\n", pic);
        return;
    }
    if (scrap_dirty)
        Scrap_Upload ();

    if ( ( ( gl_config.renderer == GL_RENDERER_MCD ) ||
          ( gl_config.renderer & GL_RENDERER_RENDITION ) ) &&
        !gl->has_alpha)
        qglDisable (GL_ALPHA_TEST);

    GL_Bind (gl->texnum);
    qglBegin (GL_QUADS);
    qglTexCoord2f (gl->sl, gl->tl);
    qglVertex2f (x, y);
    qglTexCoord2f (gl->sh, gl->tl);
    qglVertex2f (x+gl->width, y);
    qglTexCoord2f (gl->sh, gl->th);
    qglVertex2f (x+gl->width, y+gl->height);
    qglTexCoord2f (gl->sl, gl->th);
    qglVertex2f (x, y+gl->height);
    qglEnd ();

    if ( ( ( gl_config.renderer == GL_RENDERER_MCD ) ||
          ( gl_config.renderer & GL_RENDERER_RENDITION ) ) &&
        !gl->has_alpha)
        qglEnable (GL_ALPHA_TEST);
}

```

1. ábra. A Quake2 **Draw\_Pic()** függvénye (gl\_draw.c)

# 1. fejezet

## Áttekintés a ScriptKernel nyelvről

Ebben a fejezetben röviden ismertetem az általam tervezett és fejlesztett programozási nyelvet, amely a *ScriptKernel* nevet viseli [Bar13]. Ez a nyelv kezdetben szkriptelési célokra lett tervezve, de ahogy fejlődött, egyre inkább az általános célú felhasználás felé tolódott.

A ScriptKernel nyelv tervezésekor sok ötletet merítettem különböző nyelvekből. A C++ nyelv [Str13, Str94] alapján készült a szintaxis nagyobb része. A szintaxishoz hozzájárult az Ada nyelv [Bar98] néhány konstrukciója, mint például a kivételek leírásának módja (a kivételt a saját nyelvemben is „kiváltani” kell, úgy mint az Ada-ban is, nem pedig „dobni”, mint a C++ és ehhez hasonló nyelvek esetén, mint a C# [AA10] vagy a Java [Blo08]). Szintén az Ada-ból származik a kivételek kezelésének módja is: az Ada-ban csak blokk utasítás végén lehet kivételkezelő utasításokat írni. A saját konstrukcióm azonban ennél kicsit szigorúbb, és egyben hasonlít az Eiffel nyelv [Mey97] kivétel kezelési mechanizmusára – azzal a különbséggel, hogy a saját nyelvem nem teszi lehetővé az alternatív végrehajtási módok kezelését (lásd az Eiffel **retry** kulcsszavát). Az Eiffel-ben megismert elő- és utófeltétel ellenőrzés elnyerte tetszésemet, és sikeresen beépítettem a saját nyelvembe

is<sup>1</sup>.

A szintaktikai rugalmatlanságok kezelésére a privát rekord és az egyedi operátorok felhasználása segít, mert a rekordok használatával a DSL nyelv erősen típusos lesz, ami a futási idejű hibák számát redukálja, valamint „kézzel” nem lehet a reprezentált értéket megváltoztatni. Az egyedi operátorok szerepköre a DSL tevékenység leírásának megkönnyítése, és karbantarthatóságának növelése. A DSL programok írása során nem kell a kódot fölösleges függvényhívásokkal bonyolítani, mert egy jól felépített operátor precedencia és asszociativitás rendszerrel lényegretörően ki lehet fejezni a tevékenységet.

A különböző típusok és operátorok segítségével pedig olyan típusszabályrendszer építhető ki, amellyel akár a DSL problémaszintjén lehet validálni az elkészült programot.

A ScriptKernel nyelvvel készített beágyazott domain specifikus nyelvek integrálhatósága C és C++ nyelvekbe egyszerű, mert a lefordított program egy virtuális gépen fut, melyet könyvtári szolgáltatásként is lehet működtetni. Így a futtatott programot akár a gazda program leállítása nélkül is lehet cserélni. A virtuális gép egy lefordított bytekódot futtat, amit nem kell futtatás közben fordítani és ellenőrzéseket végezni, mert azok legnagyobb részét a fordító már elvégezte, így a futtatás többletköltsége nem túl magas. Továbbá a lefordított program (amennyiben a DSL típusai és operátorai megfelelően voltak felépítve) helyes, és nincs szükség rengeteg futási idejű ellenőrzésre.

A lefordított program egy saját fejlesztésű virtuális gépen fut. A virtuális gép használata lehetővé teszi, hogy a lefordított program különböző operációs rendszereken és architektúrákon is működhessen.

## 1.1. Nyelvi elemek

A ScriptKernel nyelv egy imperatív programozási nyelv, szintaxisa a C nyelvhez hasonlít. A forráskód blokkokból áll, melyek deklarációkat

---

<sup>1</sup>Az elő- és utófeltételek leírásának módja különbözik az Eiffel-ben látottól, de ez a leírás jobban illeszkedik a saját nyelvem szintaxisához (ld. A.1.6. fejezet).

és/vagy utasításokat tartalmazhatnak. Mind a blokkok, mind a tartalmuk leírásának sorrendje fontos, mert a fordító (a C nyelvhez hasonlóan) egymenetes, és csak azokat a deklarációkat ismeri, amelyek az aktuális pozícióig szerepeltek.

A szkriptnyelvek jellemzően dinamikus típusrendszerrel rendelkeznek, azonban a saját nyelvem erősen típusos, és a rengeteg hiba okát képező implicit típuskonverziót nem támogatja a nyelv. Ez bizonyos esetekben kényelmetlennek tűnhet, de az így kapott egyértelműség (függvény túlterhelés esetén) és típusbiztonság kárpótolja a programozót.

Mivel a nyelv kezdetben szkriptelési célokra lett tervezve, ezért fontos volt az is, hogy az operátorokat egyszerűen lehessen túlterhelni. Ennek megvalósítása az Ada nyelv [Bar98] operátor túlterhelési mechanizmusát használtam fel. A megvalósítás alapja az, hogy az operátorok nem speciálisabbak a függvényeknél – a különbség csak annyi, hogy az operátorokat lehet infix módon is használni.

Egyedi operátorok bevezetése funkcionális nyelvekben természetes dolog, de egy imperatív nyelvben ez egy ritka tulajdonság. Pedig egy ilyen lehetőség nagyon hasznos, főleg akkor, ha DSL-eket implementálunk. Az egyedi operátorok használatával a DSL és az eredeti probléma-térben használt jelölések nagyon hasonlóak lesznek, és ezzel növelhető az elkészített DSL átláthatósága.

A kivételkezelés is betartja a szigorú szintaktikai és szemantikai szabályokat a forráskód olvashatóságának megőrzése érdekében. Ezért a kivételkezelőket csak a függvény végén lehet elhelyezni, hogy ne keveredjen a tervezett végrehajtási út kódjával (*ld. A.1.5. fejezet*).

A nyelvi elemek kiválasztásakor elsődleges szempont volt a szintaktikai és szemantikai biztonság. Erre például szolgál az értékadás, mint utasítás. Egyes nyelvekben (például a C++) az értékadás ugyanolyan kifejezés, mint egy egyszerű összeadás. Ez egy rugalmas lehetőség, de nagyon könnyű nehezen felderíthető hibát véteni vele. A nyelv biztonságát növeli az is, hogy a kifejezéseket (így a függvények visszatérési értékeit is) kötelező felhasználni. Ennek a megszorításnak az alapja egy megtörtént eset volt: Egy kétparaméteres eljárás hívásból véletlenül ki

lett törölve az eljárás neve, de ezt a C++ fordító lefordította, nyilvánvalóan nem az elvárt funkcionalitással. Egy fél napos hibakeresés után derült fény az eltűnt azonosítóra. Ezek után ki tudná megmondani, hogy hány ilyen jellegű hibát lehetne találni nagyobb rendszerekben? Azonban, ha a nyelv nem engedi az ilyen hibák előfordulását, akkor már a fordítás során fény derült volna az előbb részletezett hibára.

Továbbá fontos szempont volt a nyelvi elemek kiválasztásakor, hogy azok lehetőleg ne interferáljanak egymással és szintaktikusan elkülöníthetők legyenek. Előnye, hogy később könnyen módosítható marad a nyelv. A jól elkülöníthető nyelvi elemek másik előnye, hogy a véletlen elgépelések esetén nem fordítható forráskód keletkezik, így a véletlen elgépelésekből nem származik futási idejű probléma.

## 1.2. Típusok

A ScriptKernel nyelv legnagyobb erőssége az erős statikus típusrendszer, melyet az implicit típuskonverziók hiánya teszi a lehető legbiztonságosabbá. Típust változtatni csak konvertáló függvényekkel lehet. A nem megengedhető konverziók (például egy negatív számot nem lehet előjel nélküli számmá konvertálni) esetén a konvertáló függvény **ConstraintErrorException** kivételt vált ki.

A nyelvben található konstans literálok típusa is egyértelmű, hogy ne okozzon félreértéseket. A C és C++ nyelvben is lehet utótagot adni a számkonstansoknak, de az csak opcionális jellegű. A ScriptKernel nyelvben az utótagot (*suffix*) kötelező megadni, és annak hiánya más típusú konstanst jelent. Például a C nyelvben a **1 == 1u** kifejezés értéke igaz, míg a ScriptKernel nyelvben ugyanez a kifejezés fordítási hibát okoz. Ennél furcsább a helyzet a **-1 == 0xffffffffu** kifejezés esetén, mert a C nyelvben ez igazra értékelődik ki, mert az összehasonlítást a bitminta alapján végzi – a ScriptKernel nyelvben természetesen ez is fordítási hiba.

A rekord a direktszorzat típuskonstrukció megvalósítása a

ScriptKernel nyelvben. A rekord típussal és névvel ellátot mezőket tartalmaz. A rekordokat lehet importálni és exportálni is. A virtuális gép az exportált rekordokat eltárolja a futtatási környezetben, importálás esetén pedig ellenőrzi a rekord nevét, mezőinek számát és típusát. (A rekord felépítését az A.14. ábrán látható BNF leírás szemlélteti.)

Bármely alaptípusból és rekordból készíthető sorozat. A típus megadása: **seq<T>**, ahol **T** bármely alaptípus vagy rekord. A nyelvben alkalmazott sorozat típus a programozási módszertanból [Fó05] ismert sorozat mintájára készült, így a használatukhoz szükséges műveletek is rendelkezésre állnak.

Az összetett típusok (rekord, sorozat, karakterlánc) paraméter átadása alapértelmezetten konstans referenciaként történik. Ez megváltoztatható a **mutable** módosító segítségével, melynek hatására a paraméterül kapott objektumok tartalma megváltoztatható (*ld. A.2.4. fejezet*).

## 2. fejezet

# A domain specifikus nyelvek megvalósítását támogató eszközök

A DSL célú felhasználás során nagy segítséget jelent a nyelv eszköztára, amely a megfelelő absztrakciós szint programozásában is segít. A fontos nyelvi elemek közé sorolható a (privát) rekord és az egyedi operátorok létrehozása. Domain specifikus nyelvek fejlesztése során komoly előnyt jelent a nyelv erős statikus típusrendszere, melynek segítségével a programozási hibák többsége fordítási időben detektálható.

### 2.1. Privát rekordok

A beágyazott DSL (*eDSL*) nyelvet és a benne írt programot célszerű két részre bontani, hogy kényelmes legyen a nyelv újrafelhasználása. A privát rekordok (*ld. A.2.2. fejezet*) használatával a DSL nyelv fejlesztését és a (korai) felhasználását szét lehet választani, mert ez a mechanizmus egy nagyon laza kapcsolatot teremt a betöltendő ScriptKernel programok között. A privát rekord segítségével a típusok felépítése teljes mértékben elrejtethető a felhasználó elől, valamint a bináris kompatibilitást is garantálja: a privát rekord importálása során a betöltő csak a

rekord nevét egyezteti, így ha a rekord felépítése megváltozik, az nem érinti a betöltött programot. A privát rekordhoz tartozó három alapműveletet a függvény importálási mechanizmuson keresztül kapja meg a program. A felhasználó programban a privát rekord deklarációjakor egyetlen mezőt sem lehet megadni, ezért a mezőkhöz közvetlen hozzáférése nincs is a felhasználó programnak. A privát rekordok egyedi típusok, melyekhez a fordító nem készít konvertáló függvényeket, ezért a privát rekordok használata egyértelműen típusozható program. A típusok helyes felépítésével olyan szabályrendszer jön létre, amivel a statikusan típushelyes program nem működik rosszul [Csö07].

```
import { record MyPrivate = private; }
import { function foo(mutable MyPrivate a); }
import { function bar(MyPrivate a); }
main
{
    MyPrivate my;
    foo(my);
    bar(my);
}
```

### 2.1. ábra. A privát rekord használata

A privát rekord használata szintaktikailag biztonságos, mert a felhasználó program nem fér hozzá a mezőkhöz, és binárisan kompatibilis is, mert betöltés során (a privát rekord nevén kívül) semmilyen paraméternek nem kell megfelelnie a betöltött programnak.

A 2.1. ábrán látható példa a privát rekord használatát mutatja be. A jelenlegi verzióban<sup>1</sup> a privát rekord és a függvény importálását külön **import** blokkba kell írni. A privát rekord bármilyen paraméter (**mutable** és nem-**mutable**) és visszatérési érték lehet. A 2.1. ábra példájához tartozó exportálások és definíciók a 2.2. ábrán láthatóak.

<sup>1</sup>ScriptKernel Compiler v0.8.2.1

```

export { record MyPrivate = private; }
export { function foo(MyPrivate a); }
export { function bar(MyPrivate a); }
declare
{
    record MyPrivate { i32 i; string s; }
}
function foo(mutable MyPrivate a)
{ a.i, a.s = a.i+1, "foo called "+tos(a.i+1)+" times"; }
function bar(MyPrivate a) { write(a.s); }
main { /* ... */ }

```

2.2. ábra. A privát rekord exportálása és definíciója

## 2.2. Operátorok

A ScriptKernel nyelv számos beépített operátor osztályt tartalmaz. Ezen osztályok közül kettő kivételével az összes túlterhelhető. A két nem túlterhelhető operátor a lusta „és” és a lusta „vagy” operátor. Mindkettő infix (bináris) operátor, és mindkét oldalon csak **bool** típusú kifejezés állhat; az operátor eredménye **bool** típusú<sup>2</sup>.

A nyelv az operátorokat egy hármas segítségével azonosítja – ezt a hármaszt a nyelv „operátor osztálynak” nevezi. Az operátorokat tartalmazó programok betöltésekor a hármas minden elemét megvizsgál, és importáláskor összeveti a virtuális gép környezetében tárolt információkkal. A hármas a következő összetevőkből áll:

1. az operátor szimbóluma (amit infix alakban is lehet használni);
2. az operátor asszociativitása (ez csak **left**, vagyis bal asszociatív, illetve **right**, azaz jobb asszociatív lehet<sup>3</sup>);
3. az operátor precedenciája (3 és 15 közötti szám, azt az operátort

<sup>2</sup>A lusta „és” operátor szimbóluma: **&&&**; a lusta „vagy” operátor szimbóluma: **|||**.

<sup>3</sup>A **left** és a **right** nem kulcsszavak, továbbra is használhatóak azonosítóként.

értékeli ki legelőször, amelyiknek a legmagasabb a precedenciája).

Az operátor szimbóluma a következő karakterekből állhat: \, -, ^, =, !, <, >, +, &, %, ~, \*, /. Megkötés, hogy nem tartalmazhatja a /\*, a \*/ és a // szekvenciák egyikét sem (ezek a megjegyzések nyitó és csukó jelei).

A nyelv beépített túlterhelhető operátorai a 2.3. ábrán láthatók. A felsorolt operátorok mindegyike bal asszociatív.

| Precedencia | Operátorok   |
|-------------|--------------|
| 5           |              |
| 6           | &&           |
| 7           |              |
| 8           | ^            |
| 9           | &            |
| 10          | ==, !=       |
| 11          | <, >, <=, >= |
| 12          | <<, >>       |
| 13          | +, -         |
| 14          | *, /, %      |

2.3. ábra. A nyelv beépített operátorai

A nyelv lehetővé teszi az egyedi operátorok létrehozását is, amely az új operátor osztályok bevezetésével történik (ld. A.1.2.4. fejezet). A 2.4. ábrán látható egy operátor osztály létrehozása. Miután az operátor osztály deklarálásra került, ugyanúgy lehet használni az egyedi operátort, mint a beépítetteket: lehet infix operátorként alkalmazni, a túlterhelés ugyanúgy történik, valamint ugyanazok az importálási és exportálási szabályok érvényesek rá. Egyetlen megkötés, hogy az operátor osztályt ugyanúgy kell deklarálni mint az operátort. Ez azt jelenti, hogy ha az új operátor csak lokális használatra készül, akkor az operátor osztály deklarációját a **declare** blokkba kell írni; ha az operátor exportálásra kerül, akkor az operátor osztálynak az **export** vagy az **import** blokkban kell lennie (attól függően, hogy egy másik program exportálta-e már az operátor osztályt – de a virtuális gép megengedi ugyanannak az operátor osztálynak a többszörös exportálását is, amennyiben az operátor

osztály mindhárom paramétere egyezik); ha az operátor (függvény) importált, akkor az operátor osztálynak is az **import** blokkban kell lennie.

```
declare { operator /= left 10; }
declare { function "!="(i32 lhs, i32 rhs) : bool; }
function "!="(i32 lhs, i32 rhs) : bool
{ return lhs != rhs; }
main
{
    write(tos( 1 /= 2 ));
}
```

#### 2.4. ábra. Példa operátor osztály létrehozására

A 2.4. ábrán egy lokálisan használató **!=** operátor látható, melynek csak egy példánya van (a **function "!="(i32 lhs, i32 rhs) : bool;**). Bármely<sup>4</sup> operátornak több példánya (túlterhelt változata) lehet, nincs megkötés a maximálisan létrehozható függvények számára.

```
declare { function almost_equal(f32 lhs, f32 rhs) : bool; }
function almost_equal(f32 lhs, f32 rhs) : bool
{ return abs(lhs - rhs) < .001f; }
main
{
    write(tos( almost_equal(10.f, 9.9995f) ));
}
```

#### 2.5. ábra. A „majdnem-egyenlő” megvalósítása függvénnyel

A következő példa már összetettebb funkcionalitást lát el. A 2.5. ábra példája egy „majdnem-egyenlő” függvény, amely két lebegőpontos számról eldönti, hogy elég közel állnak-e egymáshoz. Ugyanerre a feladatra mutat egy sokkal átláthatóbb megoldás a 2.6. ábra, ahol a megoldáshoz egy új operátor osztályt vezet be. Ennek a megoldásnak az előnye, hogy szemmel láthatóan egyszerűbb a kód és csak a lényegre

---

<sup>4</sup>A lusta „és” és a lusta „vagy” operátorok kivételével.

```

declare { operator ~= left 10; }
declare { function "=="(f32 lhs, f32 rhs) : bool; }
function "=="(f32 lhs, f32 rhs) : bool
{ return abs(lhs - rhs) < .001f; }
main
{
    write(tos( 10.f == 9.9995f ));
}

```

2.6. ábra. A „majdnem-egyenlő” megvalósítása új operátor osztály létrehozásával

koncentrálnak, valamint nem zavarják meg az olvasót a fölösleges zárójelek.

```

function mul(i32 a, i32 b) : i32
{ return a * b; }
function add(i32 a, i32 b) : i32
{ return a + b; }
main
{ i32 r = add(3, mul(4, 5)); }

```

2.7. ábra. Precedenciák szimulálása függvényhívásokkal

Az egyedi operátorokat fel lehet használni szükségtelen zárójelek és függvény hívások helyettesítésére is az operátor precedencia helyes beállításával. Így a forráskód tömörebb lesz, és a lényeges tevékenység kerül a középpontba. A 2.7. ábra példája a precedenciákat a függvényhívások szervezésével oldja meg. Ezzel szemben a 2.8. ábra példája ugyanezt a megfelelő egyedi operátorok precedenciájával oldja meg. A 2.8. ábra kódja látszólag hosszabb, de ez nem sokkal nagyobb forráskód esetén már megtérül, valamint a leírt program is tömörebb. Az operátorok használata átláthatóbbá teheti a kódot, ha a szimbólumok hasonlítanak a valós jelölésekre, valamint fölösleges zárójeleket spórolhatunk meg, amennyiben a precedenciák jól vannak beállítani. Azonban

```

declare { operator ** left 11; }
declare { operator ++ left 10; }
function "**"(i32 a, i32 b) : i32
{ return a * b; }
function "++"(i32 a, i32 b) : i32
{ return a + b; }
main
{ i32 r = 3 ** 4 ++ 5; }

```

2.8. ábra. Precedenciák és egyedi operátorok használata

a helyes használathoz többlet információra van szükség (ami a függvények esetén nem létezik): a precedencia.

## 2.3. Program frissítése futás közben

A mai világban egyre nagyobb szerepet kap a programok (a felhasználó számára) kényelmes frissítése, valamint a kiszolgáló programok esetén az, hogy a frissítés során a lehető legkevesebbet szüneteljen a szolgáltatás. A frissítések oka lehet a funkcionalitás bővítése, vagy egy programhiba javítása is. A ScriptKernel is virtuális gépen fut, ebből következik, hogy a betöltött programok a gazda program leállítása nélkül cserélhetőek. A frissítés menete dióhéjban: régi program törlése a virtuális gép futtatási környezetéből, majd az új program betöltése. Ráadásul a programok között függőségek is kialakulhatnak, melynek hatására a programokat nem lehet bármilyen sorrendben felszabadítani. Például exportált függvény importálása esetén az exportáló program függeni fog az importálótól.

Hasonló lehetőséget biztosít az Erlang [Lau13, JMCR96] nyelv lehetőséget biztosít a program (modul) futás idejű cseréjére. Ez úgy történik, hogy amint az új kód bekerül a virtuális gépbe, akkor a modulból történő („MFA” – *Module-Function-Arity*) hívások már az új modulból történnek, és ha az összes végrehajtási szál elhagyta a régi modult, ak-

kor a régi modul törtésre kerül.

A bináris formátum úgy lett megtervezve, hogy operációs rendszer és processzor architektúra független legyen. Ezt az előnyt a több platformos programok úgy tudják kihasználni, hogy a program frissítéseit csak egyszer kell lefordítani, mert működni fog minden virtuális gépen. Így egy frissítést akár hálózaton keresztül is át lehet másolni a végpontokra, és bármilyen további tevékenység nélkül le tudják cserélni az éppen futtatott változatot az újra. Az **skb** formátuma úgy lett kialakítva, hogy működjön *little endian* és *big endian* processzorokon is. A fordító mindkét mód előállítására képes, és a virtuális gép mindkét módút képes betölteni és futtatni.

A program frissítéseit nem feltétlenül csak bináris formátumban kaphatja meg egy végpont. Lehetőség van a virtuális gép futtatásával párhuzamosan programok fordítására is. A ScriptKernel programkönyvtár *API*-jában (*Application Programming Interface*) megtalálható a teljes fordítóprogram is. A megfelelő burkoló (*wrapper*) függvények megírásával (és ezen függvények hozzáadásával a virtuális gép futtató környezetéhez) lehetőség nyílik arra, hogy ScriptKernel nyelven írt program használhassa a saját fordítóját a futása során.

A virtuális gépbe betöltött programok által exportált függvényeket a ScriptKernel programkönyvtár segítségével le lehet kérni a virtuális gépből. A kapott érték egy függvény kezelő lesz, amelyet meg lehet hívni. Ennek a futtatása ugyanúgy történik, mintha a virtuális gép hívta volna meg, de a visszatérés után már nem a virtuális gép fog futni, hanem visszatér a hívás helyére. Fontos tudni, hogy az *API*-val lekért függvény kezelők függőséget állítanak be a programra, ezért ha a betöltött programot cserélni kell, akkor a lekért függvény kezelőket is fel kell előtte szabadítani.

Az Erlang virtuális gép minden modulból két különböző verziót képes tárolni és használni. A régi verziót akkor szabadítja fel, ha az összes hivatkozás megszűnt – vagyis akkor, ha az összes hívás befejeződött. Az *MFA* (*Module-Function-Arity*) hívások mindig az új modult fogják használni, míg a *FA* (*Function-Arity*) hívások mindig ugyanannak a modul-

nak a függvényeit. Az Erlang egy teljesen dinamikusan típusos nyelv, ezért a virtuális gépnek nem kell ellenőriznie a modul függvényeit és típusait.

Ezzel szemben a ScriptKernel nyelv egy erős statikus típusrendszert használ. A betöltött programokat a virtuális gépnek komoly ellenőrzéseknek kell alávetnie, hogy az importált és exportált függvények, típusok, kivételek és operátor osztályok megfelelnek-e az elvárásoknak – vagyis nem-e ütközik valamelyik része a virtuális gép futtatási környezetében található, hasonló elemével. Ezért egy Erlang-hoz hasonló mechanizmus (minden modulból/programból két változat fenntartása) jelenleg nem megoldható, mert a ScriptKernel-es programok a legkisebb interfész (importálások és exportálások együtt) változástól is használhatatlanok lehetnek az adott kontextusban. Tovább bonyolítja a helyzetet a privát rekordok létezése. Egy exportált privát rekord csak egy azonosító a felhasználó programban, nincsenek elvárásai a privát rekord felépítésével kapcsolatban. Azonban a felhasználó program tartalmazni fog egy konkrét rekordot (függetlenül attól, hogy nem hivatkozhat rá), de ha időközben ki lehetne cserélni a privát rekord definícióját tartalmazó programot úgy, hogy a rekord felépítése megváltozik, akkor a rekord régi verziója a program elszállását okozhatja.

Egy megoldási lehetőség lehet az, ha a program a privát rekord összes megvalósítását tartalmazza, és a fordító automatikusan jelöli („tagged”) azokat, és tudja kezelni a különböző verziókat. Azonban a zökkenőmentes működéshez szükséges lenne  $\rho$  függvényekre (reprezentációs függvény, amely egy konkrét rekordot képes absztrakt típusleírássá konvertálni). Azonban egy ilyen rendszer megtervezése és megvalósítása nem egyszerű feladat.

## **3. fejezet**

# **Domain specifikus nyelv megvalósítás ScriptKernel nyelven**

Ebben a fejezetben néhány domain specifikus nyelv megvalósítása található. A programok fontos részletei a hozzájuk tartozó magyarázattal kiemelésre kerültek.

Az A.4. fejezetben a 128 bites előjel nélküli egész számok megvalósítását találjuk. A típust privát rekordként exportálja a program, melyhez számos műveletet biztosít. A típusnak ugyan nincs típusinvariánsa, mint a 3.1. fejezet komplex számokról szóló példája, de ebben az esetben a megvalósítás részeinek (alsó- és felső 64 bit) önmagában nincs értelme. Az implementáció nagyban hasonlít a komplex számok megvalósításához – az implementáció fontos részeire a 3.1. fejezet kitér.

### **3.1. Komplex számok megvalósítása**

A komplex számok típusa nem része a ScriptKernel nyelvnek, azonban könnyen elkészíthető. Vannak olyan műveletek, amelyek a valós számokra (komplex esetben: a képzetes rész 0) nincs hatással. Mivel ezek a műveletek fontosak a komplex számokkal történő műveletek során,

ezért ezeket is meg kell valósítani. A kényelmes használat érdekében operátorokat kell a műveletekhez rendelni. Viszont ezek implementálásához a beépített operátor készletet ki kell egészíteni.

A komplex számokat egy rekord segítségével ábrázoljuk. Ez a rekord két 64 bites lebegőpontos számot fog tartalmazni. A valós részt a **re** mezőben, a képzetes részt az **im** mezőben tároljuk. A rekord a 3.1. ábrán látható. Ez a típus nem tartalmaz típusinvariánst, ezért a teljes rekordot exportálja a program. Ha a típusnak meg kellene felelnie egy típusinvarinnsnak, akkor azt privát rekordként érdemes exportálni, hogy a típus reprezentációjához csak a típus műveletei tudjanak hozzáférni, így a típus invariánsát meg tudják őrizni.

```
export
{
    record Complex
    {
        double re;
        double im;
    }
}
```

3.1. ábra. A komplex számok reprezentációja

A komplex konjugálás a **"~"** operátor segítségével végezhető el. Az operátor túlterhelhetőségét kihasználva a **function "~(Complex c) : Complex;** függvényt exportálja a program (az exportált függvények listája a 3.6. ábrán megtalálható). Az operátor megvalósítása a 3.2. ábrán látható.

A program exportálásai között megtalálható egy ellenőrzés vizsgálat (a megvalósítás menete hasonlít a komplex konjugáláséhoz) és egy „komplex konjugáltak-e” művelet. Ez utóbbi a **"=^="** operátorral használható (a **^** a komplex konjugáltat jelöli). Ehhez szükség van egy új operátor osztályra, mely az egyenlőség vizsgálatához hasonlóan 10-es precedenciájú bal asszociatív művelet. Ahhoz, hogy az operátort importálni lehessen, szükséges az operátor osztály exportálása (ld. 3.3. ábra),

```
function "~"(Complex c) : Complex
{
    Complex x;
    x.re, x.im = c.re, -c.im;
    return x;
}
```

3.2. ábra. A komplex konjugálás művelete

ezáltal a megfelelő precedencia és asszociativitás is garantált.

```
export
{
    operator ^= left 10;
}
```

3.3. ábra. Operátor osztály exportálása

Az operátor osztály bevezetése után a művelet implementációja már rutinszerű: a 3.4. ábrán látható kód hasonlít a már bemutatott operátor implementációhoz. Természetesen az operátor megvalósítást is exportálni kell (ez is megtalálható a 3.6. ábra exportált függvényei között).

```
function "=="(Complex lhs, Complex rhs) : bool
{
    return lhs.re==rhs.re &&& -lhs.im==rhs.im;
}
```

3.4. ábra. A „komplex konjugáltak-e” művelet

Az osztás művelet kapcsán egy érdekességet láthatunk. A 3.5. ábrán látható függvénynek van egy előfeltétele megjegyzésként. Az ugyan igaz, hogy csak akkor értelmes az osztás művelet, ha az osztó nem nulla, de ha előfeltétel nem megjegyzés lenne, és az osztó 0, akkor egy **PreconditionViolationException** kivétel váltódna ki. Ez a kivé-

```
function "/"(Complex lhs, Complex rhs) : Complex
//pre { rhs.re!=0.0 && rhs.im!=0.0 }
{
    double len = rhs.re*rhs.re + rhs.im*rhs.im;
    Complex x;
    x.re, x.im = (lhs.re*rhs.re + lhs.im*rhs.im)/len,
        (lhs.im*rhs.re - lhs.re*rhs.im)/len;
    return x;
}
```

3.5. ábra. Az osztály művelet

tel nem sugallja azt, hogy nullával való osztás történt. Ha az előfeltételt figyelmen kívül hagyjuk, akkor a **len** változóval való osztás során ki fog váltódni az elvárt **DivisionByZeroException** kivétel. Vegyük észre, hogy ennek a kivételnek a kiváltódásáig már több művelet is végre lett hajtva. Ez azonban nem tűnik túl nagy áldozatnak, hiszen a nullával való osztás egy kivételes (hibás) esemény, ezért ez a számítási többlet elfogadható.

```
export
{
    function "~"(Complex c) : Complex;
    function "=^="(Complex lhs, Complex rhs) : bool;
    function "=="(Complex lhs, Complex rhs) : bool;
    function "+"(Complex lhs, Complex rhs) : Complex;
    function "-"(Complex lhs, Complex rhs) : Complex;
    function "*" (Complex lhs, Complex rhs) : Complex;
    function "/"(Complex lhs, Complex rhs) : Complex;
}
```

3.6. ábra. Exportált függvények

A program teljes forráskódja megtalálható az A.3. fejezetben.

## 3.2. OpenGL

Az OpenGL alapvetően egy szabványt takar, melyet a különböző hardver gyártó cégek és a programzók is felhasználnak. E szabvány célja, hogy az OpenGL *API*-ban található függvények dokumentálva legyenek, valamint az, hogy a hardver gyártók ennek megfelelően tudják elkészíteni a saját termékükhöz az OpenGL *driver*-t. Az OpenGL *API*-n keresztül a programozók majdnem közvetlenül hozzáférnek a grafikus vezérlőhöz. Mivel ez tekinthető egy hardver absztrakciós rétegnek is, így az OpenGL segítségével (helyesen) elkészített programok bármely grafikus vezérlővel felszerelt számítógépen futtathatók. Az OpenGL a platform függetlenséget a C nyelvhez hasonlóan forráskód szintjén biztosítja – az OpenGL *API* egy C nyelvű programkönyvtár.

Az OpenGL-lel történő háromdimenziós kirajzolás során sok szabályra kell odafigyelni. Ezeket a szabályokat az OpenGL szabvány rögzíti. Egy OpenGL-es programot akkor tekintünk helyesnek, ha a szabványban rögzített összes kritériumnak megfelel. Ezeknek a bonyolult szabályok betartására készíthetünk egy domain specifikus nyelvet is, amely olyan interfészt biztosít a felhasználója számára, mellyel csak „szabványos” OpenGL programok készíthetők.

```
export
{
    record Vertex3f
    {
        float x;
        float y;
        float z;
    }
}
```

3.7. ábra. A csúcspont típus

A példaprogram csak a legkritikusabb részére tér ki, nem célja a teljes OpenGL programkönyvtár lefedése. A példa alapja a **glBegin()**

```

export
{
    record Color3f
    {
        float r;
        float g;
        float b;
    }
}

```

3.8. ábra. Az RGB szín típus

és a **glEnd()** hívásokkal történő háromdimenziós alakzatok kirajzolása. Az alakzatok (**Mesh**) háromszögekből és négyszögekből állnak, minden csúcspont (**Vertex3f**, lásd 3.7. ábra) egy háromdimenziós pont, és minden csúcsponthoz tartozik egy RGB szín (**Color3f**, lásd 3.8. ábra) is. A színeket lebegőpontos számokkal ábrázoljuk, a legalacsonyabb intenzitás a **0.f**, a legmagasabb pedig az **1.f**.

```

export
{
    record Triangle
    {
        bool visible;

        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Color3f  c0;
        Color3f  c1;
        Color3f  c2;
    }
}

```

3.9. ábra. A háromszög típus

Az alakzat típust privát rekordként exportálja a program, mert az

```

export
{
    record Quad
    {
        bool visible;

        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Vertex3f v3;
        Color3f  c0;
        Color3f  c1;
        Color3f  c2;
        Color3f  c3;
    }
}

```

3.10. ábra. A négyszög típus

alakzat ismeri a saját befoglaló gömbjének sugarát. Ezt a sugarat az alakzathoz hozzáadott háromszögekből (**Triangle**, lásd 3.9. ábra) és négyszögekből (**Quad**, lásd 3.10.) számolja ki. Ezért a rekordot nem lehet közönséges rekordként exportálni, mert akkor az invariáns bármikor megsérthető anélkül, hogy a program értesülne erről. Az exportált és a tényleges rekord a 3.11. ábrán látható.

Az alakzat tartalmazhat nem látható primitíveket (háromszög vagy négyszög). Ezeket a befoglaló gömb sugarába beleszámolja, mert később még láthatóvá tehetők. Az alakzathoz kétféleképpen lehet primitíveket hozzáadni: az egyik lehetőség a **Triangle** és a **Quad** típusok használatával, a másik pedig a csúcspontok koordinátáinak és színeinek megadásával. Kezdetben nem látható primitíveket csak az előbbi lehetőséggel lehet hozzáadni.

A program által exportált függvények segítségével alakzatokat lehet létrehozni és megjeleníteni. Annak érdekében, hogy egyetlen háromszög megjelenítéséhez ne kelljen egy teljes alakzatot létrehozni, az

```

export
{
    record Mesh = private;
}

declare
{
    record Mesh
    {
        seq<Triangle> tris;
        seq<Quad>      quads;
        float         radius;
    }
}

```

3.11. ábra. Az alakzat rekord

exportált függvények között található primitív kirajzoló függvény is. Az alakzat befoglaló gömbjének sugarát is egy függvény segítségével lehet lekérdezni. Az összes exportált függvény a 3.12. ábrán látható.

Ez a példaprogramban négy OpenGL-es függvényt használ a primitívek megjelenítéséhez. A program feltételezi, hogy ezek már hozzá lettek adva a virtuális gép futtatási környezetéhez, ezért ezeket a 3.13. ábra szerint importálja.

A háromszögek és négyszögek megjelenítése az OpenGL **glBegin()** és **glEnd()** hívások között elhelyezett megfelelőszámú csúcspont függvényekkel történik. A csúcspontokhoz a színt és egyéb, ebben a példában nem használt tulajdonságokat (normálvektor, textúra koordináták, stb) a csúcspont függvény előtt kell megadni. Fontos, hogy a **glBegin()**-**glEnd()** hívásokat nem lehet egymásba ágyazni, és nem lehet átlapolni őket – ezért célszerű blokkoknak kell tekinteni ezeket. Ha a **glBegin()** függvény a **GL\_TRIANGLES** paraméterrel lett meghívva, akkor a záró **glEnd()** hívásig hárommal osztható számú **glVertex{234}{sifd}()** (példánkban **glVertex3f()**) hívásnak kell lennie. Három egymást követő csúcspont függvény fog egy háromszöget alkotni. A **GL\_QUADS** pa-

```

export
{
    function AddTriangle(mutable Mesh m, Triangle t);
    function AddTriangle(mutable Mesh m,
        Vertex3f v0, Vertex3f v1, Vertex3f v2,
        Color3f c0, Color3f c1, Color3f c2);
    function AddQuad(mutable Mesh m, Quad q);
    function AddQuad(mutable Mesh m,
        Vertex3f v0, Vertex3f v1, Vertex3f v2, Vertex3f v3,
        Color3f c0, Color3f c1, Color3f c2, Color3f c3);
    function BoundingRadius(Mesh m) : float;
    function Render(Mesh m);
    function Render(Triangle tri);
    function Render(Quad quad);
}

```

3.12. ábra. Exportált függvények

raméteres eset hasonló, azzal a különbséggel, hogy nem hárommal, hanem négygel osztható darabszámú **glVertex** hívásnak kell lennie – ezek a négyes csoportok alkotják a négyszögeket. Míg a háromszög egyértelmű, a négyszög csúcspontjait úgy is meg lehet adni, hogy ne legyenek egy síkban. Az OpenGL képes megjeleníteni azokat a négyszögeket is, amelyek csúcspontjai nem egy síkot alkotnak. Ebben az esetben két háromszöget fog megjeleníteni az OpenGL úgy, hogy a két háromszög közös éle a négy megadott pont közül kettőt összeköt.

A program **function Render(Mesh m);** függvénye jeleníti meg az elkészített alakzatot. Az implementáció a nem látható primitíveket a **foreach** ciklus **which** szűrőfeltétele segítségével szűri ki: **foreach(tri in m.tris which tri.visible) { ... }**. A ciklusmag az elvárásoknak megfelelő számú **glVertex** hívást tartalmaz, így a végeredmény osztható lesz hárommal, illetve négygel.

A program teljes forráskódja megtalálható az A.6. fejezetben. A programhoz készült egy tesztprogram is, mely az A.7. fejezetben olvasható.

```

import
{
    function glBegin(u32 mode);
    function glEnd();
    function glVertex3f(f32 x, f32 y, f32 z);
    function glColor3f(f32 r, f32 g, f32 b);
}

```

3.13. ábra. Importált függvények

### 3.3. Petri-háló

A nyelv DSL képességeinek tesztelésére készítettem egy Petri-hálókkal [Pet62] foglalkozó program könyvtárat. A Petri-háló egy páros gráf, a *helyeket* és az *átmeneteket* irányított élek kötik össze. Az átmenetek a helyek súlyozását (a súly egy természetes szám) módosítják: amelyikből mutat ki él, annak a súlyozását csökkenteni kell az élen szereplő értékkel (ha nincs megadva, akkor 1); amelyikbe mutat él, akkor a súlyozását pedig növelni kell. Egy átmenet akkor hajtható végre, ha az átmenetet megelőző helyek súlyozása nem kisebb, mint a helyet és az átmenetet összekötő él súlya.

Az elkészített programmal szabályos Petri-hálót lehet építeni, melynek helyességét fordítási időben a fordító ellenőrzi. Az elkészített hálókkal szimulációkat is lehet végezni. A szimuláció végrehajtásának helyességét futási időben ellenőrzi, és hibás átmenet végrehajtásakor **PNWeightErrorException** kivételt kapunk. A hálók építéséhez és végrehajtásához használt típusok, kivételek és függvények egy különálló programot (`petri.skb`) alkotnak. Ehhez tartozik egy *import include* is (`petri.import.inc`).

A Petri-háló típus egy privát rekord, melyet példányosítás után inicializálni kell. A helyeket és átmeneteket prototípusként (lehet címkéje és súlya) lehet hozzáadni a hálóhoz, és a hozzáadás után „igazi” helyek és átmenetek lesznek belőlük. Ezekből a helyekből és átmenetekből lehet relációkat létrehozni, és hozzáadni a hálóhoz. A reláció létrehozó

operátorok úgy vannak elkészítve, hogy átmenetet csak hellyel, valamint helyet csak átmenettel lehet összekapcsolni – ellenkező esetben fordítási hibát kapunk.

```
import
{
    record PetriNet = private;

    record PNTrans = private;
    record PNPlace = private;
    record PNTransProto = private;
    record PNPlaceProto = private;

    record PNRelationP = private;
    record PNRelationT = private;

    operator += left 3;
    operator <- left 4;
    operator -> left 4;
}

import
{
    function Init(mutable PetriNet pn);

    function Place() : PNPlaceProto;
    function "&&"(PNPlaceProto p, string name) : PNPlaceProto;
    function "&&"(PNPlaceProto p, u32 w) : PNPlaceProto;

    function Trans() : PNTransProto;
    function "&&"(PNTransProto t, string name) : PNTransProto;

    function "+="(mutable PetriNet pn, PNPlaceProto p) : PNPlace;
    function "+="(mutable PetriNet pn, PNTransProto t) : PNTrans;
    function "+="(mutable PetriNet pn, PNRelationT rel);
    function "+="(mutable PetriNet pn, PNRelationP rel);
    function "+="(mutable PetriNet pn, seq<PNRelationT> rels);
    function "+="(mutable PetriNet pn, seq<PNRelationP> rels);

    function "->"(PNTrans t, PNPlace p) : PNRelationP;
    function "->"(PNPlace p, PNTrans t) : PNRelationT;
    function "->"(PNTrans t, u32 w, PNPlace p) : PNRelationP;
    function "->"(PNPlace p, u32 w, PNTrans t) : PNRelationT;
    function "->"(PNRelationT rel, PNPlace p) : seq<PNRelationP>;
    function "->"(PNRelationP rel, PNTrans t) : seq<PNRelationT>;
    function "->"(PNRelationT rel, u32 w, PNPlace p) : seq<PNRelationP>;
    function "->"(PNRelationP rel, u32 w, PNTrans t) : seq<PNRelationT>;
    function "->"(seq<PNRelationT> rels, PNPlace p) : seq<PNRelationP>;
    function "->"(seq<PNRelationP> rels, PNTrans t) : seq<PNRelationT>;
```

```

function "->"(seq<PNRelationT> rels, u32 w, PNPlace p) : seq<PNRelationP>;
function "->"(seq<PNRelationP> rels, u32 w, PNTrans t) : seq<PNRelationT>;

function "+"(PNTrans t0, PNTrans t1) : PNProcess;
function "+"(PNProcess p, PNTrans t) : PNProcess;

function "<-"(mutable PetriNet pn, PNTrans t);
function "<-"(mutable PetriNet pn, PNProcess p);

function Reset(mutable PetriNet pn);

function Weights(PetriNet pn, bool labs) : string;
}

```

A hálót a **PetriNet** típus reprezentálja; a helyek prototípusa a **PNPlaceProto**, a végleges hely a **PNPlace**, az átmenetek típusai a **PNTransProto** és a **PNTrans**. A **PNRelationP** egy olyan reláció, amely átmenetből mutat helybe, a **PNRelationT** pedig helyből mutat átmenetbe.

A **Reset()** függvény visszaállítja a hálóban a létrehozáskor megadott súlyozást. Vagyis minden hely súlya felülíródik a létrehozáskor megadott súllyal.

A **Weights()** függvény létrehoz egy szöveget, amelyben felsorolja az egyes helyekhez tartozó súlyozást. Amennyiben a **labs** paraméter igaz, akkor a visszaadott szövegbe elhelyezi a helyek címkéit is.

### 3.3.1. Háló létrehozása

A következőkben egy egyszerű, 2 helyből és 3 átmenetből álló hálót létrehozó kód található. A helyek kezdetben 0 súlyúak, nevük legyen **p1** és **p2**. Az átmenetek nevei sorra **t0**, **t1** és **t2**. Élek 1 súllyal: **t0** → **p1**, **p1** → **t1**, **t1** → **p2**; él 2 súllyal: **p2** → **t2**.

A 0 súlyú **p1** hely létrehozása: a **Place()** függvénnyel lehet elkészíteni a hely prototípust, ehhez az **&&** operátorral lehet címkét és súlyt hozzárendelni. Az operátorral megváltoztatott prototípust a **+=** operátor adja hozzá a háléhoz; a **+=** operátor eredménye egy *hely*. A helyet a **p1** nevű lokális változóban tárolja el a program – ezt a helyet a relációk létrehozásakor felhasználja.

Az átmenetek létrehozása nagyon hasonló a helyek létrehozására, de a helyekkel ellentétben az átmenetekhez nem lehet súlyozást hozzárendelni.

```
main
{
    PetriNet pn;
    Init(pn);

    PNTrans t0 = pn += Trans() && "t0";
    PNTrans t1 = pn += Trans() && "t1";
    PNTrans t2 = pn += Trans() && "t2";

    PNPlace p1 = pn += Place() && "p1" && 0u;
    PNPlace p2 = pn += Place() && "p2" && 0u;

    // ....
```

A helyek és az átmenetek létrehozása után a relációkat kell létrehozni. Az éleket egyesével hozzáadja a hálózhoz, és ahol szükséges, ott az alapértelmezett 1 súly helyett más súlyt állít be. Ezt a **->** operátor három paraméteres változatával lehet elérni. Ez három különböző alakban is leírható:

1. **"->"(p2, 2u, t2);**
2. **(p2, 2u) -> t2;**
3. **p2 -> (2u, t2);**

A későbbiekben látható, hogy célszerűbb a 3. formát használni.

```
// ....

pn += t0 -> p1;
pn += p1 -> t1;
pn += t1 -> p2;
```

```
pn += p2 -> (2u, t2);  
  
// ...
```

A fenti kódrészlettel ekvivalens az alábbi:

```
// ....  
  
pn += t0 -> p1 -> t1 -> p2 -> (2u, t2);  
  
// ...
```

Ez utóbbi felhasználja azt a lehetőséget, amellyel relációkat lehet sorozatba szervezni. Végül a létrehozott reláció sorozatot adja hozzá a hálózhoz.

### 3.3.2. Háló szimulálása

A következő példa végrehajtja elkészült hálón a  $t_0, t_0, t_1, t_1, t_2$  átmenet sorozatot. Hibakereséshez jól jöhet a **Weights()** függvény, melynek segítségével kiírható a súlyozás.

```
write("M: " + Weights(pn, true) + "\n");
```

Ennek a kódrészletnek a végrehajtása után az "M: 0, 0" szöveg fog megjelenni a képernyőn.

Az alábbi átmenet végrehajtása után a súlyozás megváltozik erre: "M: 1, 0".

```
pn <- t0;
```

A fenti átmenet újbóli végrehajtása után a súlyozás "M: 2, 0"-ra változik.

Lehetőség van több átmenet végrehajtására is: az átmeneteket a **+** operátorral össze lehet fűzni átmenet sorozatba. Vagyis a célként kitűzött  $t_0, t_0, t_1, t_1, t_2$  átmenet sorozatot elő lehet állítani:

```
PNProcess proc = t0 + t0 + t1 + t1 + t2;
```

Az előállított átmenet sorozatot nem szükséges eltárolni, lehetőség van azonnali felhasználásra is. Fontos megjegyezni, hogy a `+` operátor precedenciája magasabb, mint a `<-` operátoré, vagyis először az átmenet sorozat értékelődik ki, majd utána az átmenet sorozat végrehajtása.

```
pn <- t0 + t1 + t1;
```

Az eddigi átmenetek végrehajtását figyelembe véve a jelenlegi súlyozás a következő: "M: 1, 2". A sorozat utolsó átmenetének végrehajtása után (vagyis a `t2` végrehajtása után) a súlyozás erre változik: "M: 1, 0", mert a `t2` él súlya 2 volt, így a `p2` helyről 2 súlyt vesz le.

```
pn <- t2;
```

A program forráskódja kisebb eltérésekkel megtalálható a függelékben (*ld. A.8. fejezet*).

### 3.3.3. Csokiautomata

A csokiautomata egy olyan állapotgép, amely 5 helyből (állapotból), és 9 állapotátmenetből épül fel. Az állapotok a bedobott pénz mennyiségét tárolják, az átmenetek pedig az 5 és 10 értékű érme bedobását, valamint a kis csoki (15 pénzbe kerül) és nagy csoki (20 pénz) megvásárlását szimulálják. Ez egy Petri-hálóval felírható, melynek kódja a következő:

```
PNPlace p0 = pn += Place() && "0" && 1u;  
PNPlace p5 = pn += Place() && "5" && 0u;  
PNPlace p10 = pn += Place() && "10" && 0u;  
PNPlace p15 = pn += Place() && "15" && 0u;  
PNPlace p20 = pn += Place() && "20" && 0u;
```

```

PNTrans t0_5    = pn += Trans() && "+5";
PNTrans t5_10   = pn += Trans() && "+5";
PNTrans t0_10   = pn += Trans() && "+10";
PNTrans t5_15   = pn += Trans() && "+10";
PNTrans t10_15  = pn += Trans() && "+5";
PNTrans t15_20  = pn += Trans() && "+5";
PNTrans t10_20  = pn += Trans() && "+10";
PNTrans t15_0   = pn += Trans() && "Kis";
PNTrans t20_0   = pn += Trans() && "Nagy";

```

A helyek és állapotok után az átmenetek hozzáadása már nagyon egyszerű, tömör és szemléletes:

```

pn += p0 -> t0_5 -> p5 -> t5_10 -> p10 -> t10_15 -> p15
      -> t15_20 -> p20 -> t20_0 -> p0;
pn += p0 -> t0_10 -> p10 -> t10_20 -> p20;
pn += p5 -> t5_15 -> p15 -> t15_0 -> p0;

```

Az állapotgép a  $p_0$  állapotból indul, vagyis ezen a helyen 1 súly található. A többi hely ( $p_5$ ,  $p_{10}$ ,  $p_{15}$  és  $p_{20}$ ) 0 súlyúak. Az átmenetekkel a  $p_0$  helyen található súlyt lehet mozgatni a hálóban. A hálóban található helyek súlyainak összege bármely elérhető állapotban 1, vagyis ez a Petri-háló valóban *állapotgép*.

A háló összeállítása után a szimuláció már csak a Petri-háló lekérdező műveleteit és az átmenet végrehajtását használja. Az alábbi program minden iterációban lekérdezi a végrehajtható átmeneteket (ezt az **AvailableTrans()** függvény szolgáltatja egy sorozat formájában) és ki-listázza azokat. A felhasználónak be kell írnia a végrehajtandó átmenet nevét. Ha megfelelő a beírt szöveg, akkor az átmenet végrehajtásra kerül. Ezáltal a háló súlyozása is megváltozik, így a végrehajtható átmenetek halmaza is megváltozik. A hibásan megadott átmenetről a felhasználó értesül. Kilépni az üres sorral lehet.

```

bool run = true;
while(run)

```

```

{
    seq<PNTrans> ts = AvailableTrans(pn);
    write("Available transitions:\n");
    foreach(i in 0u to seqlen(ts) exclusive)
    {
        write("\t" + TransName(ts[i]) + "\n");
    }

    string choice = read();
    if(strlen(choice)==0)
    {
        write("No input, exit...\n");
        run = false;
    }
    else
    {
        bool done = false;
        foreach(i in 0u to seqlen(ts) exclusive while !done)
        {
            if(strcmp(TransName(ts[i]), choice)==0)
            {
                pn <- ts[i];
                done = true;
            }
        }
        if(!done)
        {
            write("No such transition.\n");
        }
    }
}

```

A program teljes forráskódja megtalálható a függelékben (*ld. A.9. fejezet*).

# Összegzés

Dolgozatomban ismertettem a ScriptKernel nevű, saját fejlesztésű programozási nyelvem felépítését, szintaxisát és szemantikáját. A nyelv kapcsán bemutattam, hogy a szigorú szintaxtikai és szemantikai szabályok, mint az erős statikus típusrendszer növeli a nyelven írt program megbízhatóságát. Ehhez a programozási nyelvhez tartozó fordítóprogramot, virtuális gépet és egyéb segédeszközöket le lehet ingyenesen tölteni [Bar13].

Bemutattam, hogy ez a saját fejlesztésű programozási nyelv miként alkalmazható beágyazott domain specifikus nyelvek készítésére. Kiemeltem, és részletesen bemutattam az ide vonatkozó nyelvi elemeket, valamint a virtuális gépből adódó előnyös tulajdonságokat.

Mutattam példát a különböző DSL fejlesztési technikák felhasználására. A komplex számos példa fő célja az operátorok és egyedi operátorok bemutatása volt, míg az OpenGL-es példaprogram a domain szabályainak betartására összpontosult. A példák bemutatása a fontos részletek magyarázatával történt.

Példaprogramokon keresztül bemutattam, hogyan lehet felhasználni a ScriptKernel nyelvet beágyazott domain specifikus nyelvek készítésére. A futtatása történhet programkönyvtár segítségével, de lehetőség van különálló programként is futtatni az elkészített DSL programokat.

Ebben a nyelvben való fejlesztés segítésére készítettem egy szintaktis kiemelő leírást, melyet a *gedit* szövegszerkesztővel lehet használni. Ez a szintaktis kiemelő leírás szintén ingyenesen letölthető.

A letölthető csomagban megtalálható a fordítóprogram és a futtató program (virtuális gép) több operációs rendszerre (Linux, Windows) és

processzor architektúrára (x86, ARM CortexA8) lefordított változata. Ebben a csomagban található néhány példa program is, melyeket a fejlesztés során készítettem, és a regressziós tesztek során használok is.

A ScriptKernel nyelven írt eddigi legnagyobb programom egy közel 3500 soros 3-dimenziós pályaszerkesztő program. A lefordított program 18 000 végrehajtható utasítást tartalmaz.

## Továbbfejlesztési lehetőségek

A nyelv rugalmasságát növelné, ha lenne lehetőség generikus függvények (illetve később generikus osztályok) készítésére. Ez az új programozási paradigma sokkal tágabb lehetőségeket nyitna meg a kód újrafelhasználása terén. Generikusság alatt nem a C++ nyelv *template* mechanizmusát értem, ahol a kód újra felhasználása forráskód szinten történik. Sokkal inkább az Ada és a Java nyelvekben használt generikussághoz hasonlít, ahol a sablont külön kell lefordítani, és a behelyettesítés során nem keletkezik másolat az eredeti függvényről. Ennek az az előnye, hogy a sablon forráskódját el lehet rejteni a felhasználók elől, ami nagyobb biztonságot jelent a program számára.

Terveim között szerepel a funkcionális nyelvekből ismert algebrai adatszerkezetek beépítése is. Ezt a típuskonstrukciót több módon is fel lehet használni, például biztonságos unió típus, valamint rekurzív adatszerkezetek (például fák) létrehozására. Az adatkonstruktorok szétválasztására a C nyelv **switch-case** konstrukcióját fogom felhasználni úgy, hogy az egyes esetek az adatkonstruktor nevével kezdődik, utána az adatkonstruktor paramétereinek számával megegyező azonosító, melyekkel a tárolt értékekre lehet hivatkozni. Ezt a lehetőséget később többszálú programok esetén is alkalmazni lehet, mert ennek segítségével típusbiztosan lehet bármilyen típusú értéket átküldeni egy másik szálnak – vagy akár egy másik számítógépen, szintén a nyelvhez tartozó virtuális gépen futó alkalmazásnak.

A virtuális gép rugalmasabb használata érdekében tervezem a funk-

cionalitását bővíteni egy program frissítő (cserélő) mechanizmussal. Ez a mechanizmus ellenőrizné az új programot, hogy az általa használt interface megfelelő-e, vagyis csak akkor lehet frissíteni a programot, ha a régi program által exportált és felhasznált függvények, kivételek, típusok és operátor osztályok közül egyik sem hiányzik az új programból. Ez nem azt jelenti, hogy minden exportált objektumnak meg kell lennie az új programban, hanem azt, hogy a használtak (amit más programok felhasználnak, vagy API-n keresztül le lett kérdezve) közül mindegyik létezzen, és ugyanazok legyenek (ugyanaz a függvény szignatúrája, ugyanaz a rekord felépítése, stb.).

# Irodalomjegyzék

- [AA10] Joseph Albahari and Ben Albahari. *C# 4.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2010.
- [Bar98] John Barnes. *Programming in Ada 95*. Addison-Wesley, 2nd edition, 1998.
- [Bar12] Áron Baráth. A ScriptKernel programozási nyelv (TDK dolgozat), 2012. Témavezető: Tóth Melinda.
- [Bar13] Áron Baráth. ScriptKernel v0.8.2.1, 2013. <https://people.inf.elte.hu/baaqaai/skc>.
- [Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.
- [BP13] Áron Baráth and Zoltán Porkoláb. Domain-specific languages with custom operators. In *13th Symposium of Programming Languages and Software Tools*, page submitted, August 2013.
- [Csö07] Zoltán Csörnyei. *Lambda-kalkulus*. Typotex Elektronikus Kiadó, 2007.
- [Emm09] P. Emmerich. *Beginning Lua with World of Warcraft Add-ons*. Apresspod Series. Apress, 2009.
- [Fow10] Martin Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

- [Fó05] Ákos Fóthi. *Bevezetés a programozáshoz*. ELTE Eötvös Kiadó Kft., 2005.
- [Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM computing surveys*, 28(4es):196, 1996.
- [Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
- [IDFF96] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Software Practice and Experience*, 26(6):635–652, 1996.
- [IEE13] IEEE. Ieee standard for floating-point arithmetic, 2013. <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933>.
- [Ier06] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2006.
- [iS97] id Software. Quake 2, 1997. <http://www.idsoftware.com/games/quake/quake2>.
- [JMCR96] Armstrong J., Williams M., Wikstrom C., and Virding R. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.
- [JMN<sup>+</sup>10] Muller Jean-Michel, Brisebarre Nicolas, de Dinechin Florent, Jeannerod Claude-Pierre, Lefèvre Vincent, Melquiond Guillaume, Revol Nathalie, Stehlé Damien, and Torres Serge. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
- [Lau13] Simon St. Laurent. *Introducing Erlang*. O'Reilly Media, 2013.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

- [MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
- [OSG08] Bryan O’Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O’Reilly Media, 2008.
- [OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.
- [Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1st edition, 1994.
- [Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.
- [VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatica*, 2000.
- [Wik13] Wikipedia. Domain-specific language, 2013. [http://en.wikipedia.org/wiki/Domain-specific\\_language](http://en.wikipedia.org/wiki/Domain-specific_language).

# A. Függelék

## A.1. Nyelvi elemek

Ez a fejezet az 1.1. fejezetben használt fogalmakat mutatja be részletesen. Továbbá a ScriptKernel nyelv szintaxisát, szemantikáját és nyelvi elemeit ismerteti.

### A.1.1. Blokkok

A forráskódban található blokkokat három csoportba lehet sorolni: deklarációkat tartalmazó blokk (ez lehet **declare**, **import** és **export**), globális változókat tartalmazó blokk (ez a **global** blokk), valamint utasításokat tartalmazó blokk (ezek a **ctor**, a **dtor**, a **main** és a **function** blokkok).

A deklarációkat és a globális változókat tartalmazó blokkok tartalmát a fordító képes összevonni, vagyis ezekből a blokkokból több is lehet. Ezek a blokkok nem lehetnek üresek. A **declare**, **import** és **export** blokkok tartalmát a fordítóprogram összevonja (unió), így a többszörös deklarációk nem okoznak gondot. A **global** blokk változóinak egyedi-nek kell lennie, ezért ha ebben a blokkban névütközés történik, az fordítási hibát okoz.

Az utasításokat tartalmazó blokkokból csak egy szerepelhet a forrásban (a **function** blokk esetén szignatúránként egy szerepelhet). A **ctor** és a **dtor** blokk opcionális, de a **main** blokk megadása kötelező, és csak egy lehet belőle (a törzse akár üres is lehet). Ezen blokkok tartal-

mazhatnak utasításokat és értékadásokat (*ld. A.1.4. fejezet*), valamint változó deklarációkat (*ld. A.1.2. fejezet*) is.

A **declare** blokkba írt deklarációk (függvény, kivétel, típus, operátor osztály) lokálisak a programra nézve. Fontos szerepet játszanak a rekurzív függvények leírása során. A **export** blokk deklarációi használhatóak lokális deklarációnak is, de ezek a deklarációk a lefordított programban is szerepelnek. A virtuális gép ezen deklarációkat hozzáadja a futtató környezethez, és a később betöltött programok felhasználhatják ezeket. Az **import** blokkban szereplő deklarációk szintén használhatóak lokálisan is, de ezeket már a futtató környezettel szinkronizálja. Az importált függvények törzsét más programoktól kapja.

## A.1.2. Deklarációk

### A.1.2.1. Változók

A változó deklaráció a C nyelvhez hasonlóan épül fel. A változó típusa nyitja a deklarációt (ami egyben definíció is), és a deklarációt a pontosvessző zárja. A típus után kell felsorolni a létrehozandó változó nevét vagy változók neveit. Opcionálisan lehet kezdőértéket is adni, melyet a változólista után írt **=** vagy **:=** (értékadás) jel után kell írni. A megadott kifejezés szigorú típusellenőrzésen megy át. Ha nincs kezdő értékadás megadva, akkor minden változót a típus nullelemével inicializálja (**i32** esetén **0**, **f32** esetén **0.f**, **string** esetén **""**, stb. (*ld. A.2. fejezet*)).

```
<type> <identlist> ;  
<type> <identlist> = <expr> ;
```

### A.1.2.2. Függvények

A függvény deklarációt úgy kellett megtervezni és felépíteni, hogy egyértelműen le lehessen írni vele a több visszatérési értéket szolgáltató függvényeket. A megoldás alapját az Ada nyelvben alkalmazott

függvény szignatúra adta.

```
function ( <args> ) ;  
function ( <args> ) : <types> ;
```

#### A.1. ábra. Függvény deklaráció

Speciális eset az eljárás, mert az eljárás hívása utasításnak minősül. Az eljárások jelölése a visszatérési értékek elhagyásával történik. Az A.1. ábra első sorában látható az eljárások általános leírása. A visszatérési érték megadása a C nyelvből ismert **return** utasítás segítségével történik. Ez eljárások esetén egy „üres” *return: return;*. Ha ez az „üres” *return* az eljárás végére kerülne, akkor elhagyható, mert az alapértelmezett viselkedés (eljárások esetén) az „üres” *return*. Erre mutat példát az A.3. ábra. A függvények esetén a visszatérési érték beállítása kötelező, ezért a **return** nem hiányozhat. A **return** után, és a ; (pontosvessző) előtt a visszatérési értéknek kell állnia. A függvény visszatérési értékének beállítását illusztrálja az A.2. ábra.

```
function foo(u32 a, u32 b) : bool, u32  
{  
    return a>b, a+b;  
}
```

#### A.2. ábra. Példa egy függvényre

A függvény deklaráció ; (pontosvessző) előtti része meg kell egyezzen a függvény definíció (függvény blokk) { (nyitó kapcsos zárójel) előtti részével – azaz a függvény szignatúrájának meg kell egyeznie.

A ScriptKernel nyelvben az operátorok nem sokban különböznek a közönséges függvényektől. A legfontosabb különbség az, hogy az operátorok írhatóak infix (vagy prefix) alakban. Deklarációjuk azonban ugyanúgy néz ki, mint a közönséges függvények deklarációja, azzal a különbséggel, hogy az operátort idézőjelek (") közé kell írni.

```
function print(string s)
{
    write(s + "\n");
}
```

A.3. ábra. Példa egy eljárásra

```
function "+"(i32 lhs, u32 rhs) : u32;
```

A deklarációból adódik, hogy egy bináris operátornak az operandusai nem csak egyszeres típusok lehetnek. Egyszeres típus alatt az olyan kifejezéseket értjük, amelyeknek csak egy típusa van. A kifejezések , - vel (vessző) konkatenálhatóak, ebben az esetben a kifejezés típusa a két típus összevonásra kerül (*ld. A.1.3. fejezet*).

```
function "+"(u32 a, u32 b, u32 c) : u32
{ return a + b + c; }
// ...
u32 sum = 5u + (6u, 7u); // 18u
```

Mivel az operátorok is függvények, ezért az operátort is lehet függvényhívásként leírni. Az előző példa klasszikus függvényhívás formában:

```
u32 sum = "+"(5u, 6u, 7u); // 18u
```

### A.1.2.3. Kivételek

A kivételek csak azonosítók, nem hordoznak információt. Ezért a leírásuk, importálásuk, exportálásuk és a kivétel terjedése nem igényel komplex műveleteket. A virtuális gépbe történő betöltés során minden kivétel kap egy globálisan egyedi azonosítót, melyet a programok a lokális sorszámmal azonosítanak. A lokális kivétel azonosító cseréjének műveletigénye globális azonosítóra konstans.

Kivételt csak deklarálni (**declare** blokk) nem lehet, mert a programnak szüksége van a globális kivétel azonosító használatára. Minden exportált (a **declare** blokkba írt kivételek is exportálásra kerülnek) kivételt automatikusan importálni fog a program, ezzel garantálja a globális kivétel azonosító meglétét.

Importálni csak exportált kivételt lehet (a virtuális gép futtatási környezetéből – egy másik program által exportált kivételt), valamint a standard kivételeket.

```
exception MyException;
```

A kivételkezelők után opcionálisan a **finally** blokk foglalhat helyet. Ennek a blokknak a tartalmát a függvényből való kilépés előtt lefut (akár kivételt történt, akár **return** utasítás hatására). A **finally** után utasítás már nem szerepelhet a függvény csukó kapcsos zárójeléig. Ha mégis lenne utasítás a kivételkezelők után, az fordítási hibát okozna.

#### A.1.2.4. Operátor osztályok

A ScriptKernel nyelvben lehetőség van egyedi operátorok létrehozására is. Az új operátort csak új *operátor osztály* létrehozásával lehet bevezetni. Az operátor osztály három részből áll: operátor szimbólum (amit később infix módon lehet használni), asszociativitás (ez csak **left**, vagyis bal, valamint **right**, vagyis jobb lehet), valamint a precedencia szint (3 és 15 közötti szám; minél magasabb a precedencia, annál előbb hajtódik végre). Az operátor osztály létrehozása után az operátort megvalósító függvények a már megszokott módon adhatók meg (*ld. A.1.2.2. fejezet*).

Az operátor osztályok célja az, hogy egy adott operátor szimbólumhoz csak egy asszociativitás és precedencia tartozzon. Ezért fontos ezek importálása és exportálása. Problémát okozhat az, ha egy exportált operátort az importáló program más precedenciával vagy asszociativitással használja fel, így az operátor szemantikája meg is változhat. Az operátor

osztályokat a virtuális gép nyilvántartja, és ellenőrzi a betöltött programok importált és exportált operátor osztályait.

```
operator <symbol> <assoc> <prec> ;  
// ...  
function "<symbol>" ( <args> ) : <types> ;
```

A következő példa létrehozza a „körülbelül egyenlő” operátor osztályát, amit a `=` szimbólummal lehet hivatkozni rá. Egy operátor megvalósítást is létrehoz, amely két 32 bites lebegő pontos számról eldönti, hogy „körülbelül egyenlők-e”. Majd ezt az operátort felhasználja egy logikai változó inicializálásában.

```
operator ~= left 10;  
// ...  
function "~="(f32 lhs, f32 rhs) : bool  
{ return abs(lhs - rhs) < 0.001f; }  
// ...  
bool b = 1.0001f ~= 1.f;
```

A nem exportált egyedi operátor megvalósításhoz tartozó operátor osztályt nem kötelező exportálni. A következő példában használt `==>` operátor osztályt elég csak deklarálni (**declare** blokk), nincs szükség exportálásra. (Feltéve, hogy a **function "=="(bool a, bool b) : bool;** függvény nincs exportálva.)

```
declare  
{  
    operator ==> left 11;  
    // ...  
    function "=="(bool a, bool b) : bool;  
}
```

### A.1.3. Kifejezések

Más néven: *jobbérték*.

Kifejezés lehet literál, vagy balérték. A kifejezések a , (vessző) segítségével konkatenálhatók, így többértékű (és több típusú) kifejezések hozhatók létre (nem összetévesztendő a C++ szekvencia operátorával, mert a ScriptKernel nyelvben itt egy többértékű kifejezést ír le a vessző). A kifejezések tetszőlegesen bezárójelezhetők. Prefix operátor alkalmazása kifejezésre is kifejezés. Infix operátor alkalmazása két kifejezésre is kifejezés. A függvényalkalmazás eredménye is kifejezés.

A kifejezések BNF-je az A.4. ábrán látható.

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

A.4. ábra. A kifejezések BNF-je

### A.1.4. Értékadás

Az értékadás egy utasítás, amely egy kifejezés értékét tárolja el egy balértékben. A C és C++ nyelvvel ellentétben az értékadás nem utasítás, vagyis nem lehet (akár véletlen, akár szándékosan) elágazás feltétele. Ezzel is biztonságosabbá válik a nyelv, mert ha egy elgépelés fordítási időben kiderül, akkor könnyen javítható a hiba, de ha futás közben nem úgy viselkedik a program, mint ahogy gondoltuk (csak elírás miatt lett az egyenlőségvizsgálatból értékadás), akkor fáradtságos és költséges a hibát megkeresni a programban.

```
<lvalue> = <rvalue> ;  
<lvalue> := <rvalue> ;
```

#### A.5. ábra. Az értékadás

Értékadás jelölésére az = (egyenlő) és a := (kettőspont egyenlő) jel bármelyike használható. Az A.5. ábrán az **<rvalue>** egy jobbérték (kifejezés, lásd A.1.3. fejezet), az **<lvalue>** egy balérték.

A balértékeket is lehet , -vel (vessző) konkatenálni. Ennek hatására több értéket lehet egyidejűleg eltárolni (szimultán értékadás). Balérték lehet egy változó (esetleg a *scope* megadásával), egy sorozat eleme (*ld. A.2.3. fejezet*) és egy rekord mezője (*ld. A.2.2. fejezet*).

Az értékadás során a jobbérték teljes értékét el kell tárolni, nem maradhat eltárolatlan érték. Ez a viselkedés is biztonsági szempontok miatt került bele a nyelvbe. Az értékadás során rengeteg ellenőrzést hajt végre a fordító: a jobb- és balértékek típusainak számát és egyenlőségét is ellenőrzi. Példuál egy (**i32**, **u32**) típusú kifejezést nem lehet eltárolni egy (**i32**, **i32**) típusú balértékben, mert a típusok darabszáma egyezik, de a második típus nem. Fontos megjegyezni, hogy az **i32** és az **u32** típusok is 32 bites egészek, de a két típus értékkészlete nem egyezik meg, így egy **u32** típusú értéket nem lehet minden esetben **i32** típusúnak tekinteni – az implicit típuskonverzió hiánya pedig ezt a problémát elhárítja (explicit típuskonverzió szükséges, ami hibás konverzió esetén kivételt vált ki).

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

### A.1.5. Kivételkezelés és -kiváltás

A kivételeket csak a függvény blokk végén lehet kezelni. Így a kivételkezelés nem fog összefolyni a normál viselkedés kódjával (a kivétel egy nem várt, nem normális eseményt jelent), ezáltal a kód olvashatóbbá válik. Lehetőség van egy adott kivételt lekezelni (a kivétel azonosítójának megadásával), valamint minden (le nem kezelt) kivételt lekezelni – ez utóbbi csak utolsó helyen szerepelhet.

```
main
{
    // ...

    exception(OutOfMemoryException)
    {
        // ...
    }
    exception(IndexOutOfRangeException)
    {
        // ...
    }
    exception(...)
    {
        // ...
    }
}
```

A fenti példában a harmadik kivételkezelő minden kivételt lekezel, kivéve a **OutOfMemoryException** és a **IndexOutOfRangeException** kivételeket.

Kivételt kétféleképpen lehet kiváltani. Az egyik módja az, hogy a kivételt a hívó függvény kezdi el lekezelni (**raise**), a másik módja az, hogy a kivételt kiváltó utasítást tartalmazó függvény kezdi el lekezelni

(**raise near**<sup>1</sup>). Ez utóbbi felhasználható a vezérlési szálba való beavatkozáshoz.

```
function foo()
{
    raise(MyFarException);
    raise near(MyNearException);

    exception(MyNearException)
    {
        // ...
    }
}
```

#### A.1.6. Elő- és utófeltételek

A ScriptKernel nyelv támogatja a függvények elő- és utófeltételeinek ellenőrzését is. A fordítás során ezeket a feltételeket és ellenőrzéseket ki is lehet hagyni a végleges programból (egy végleges termékben nem feltétlen szükségesek). Ezeknek az ellenőrzése automatikus, nem kell „kézzel” ellenőrizni őket. Ha az előfeltétel nem teljesül, akkor **PreconditionViolationException** kivételt fog dobni (mivel ebben a pillanatban logikailag még nem lépett be a függvénybe, így a függvény saját kivételkezelő része nem hajtodik végre). Az utófeltétel sérülésekor **PostconditionViolationException** kivételt vált ki.

Az előfeltétel megadása a **pre** kulcsszó segítségével történik. Az előfeltételben (amelynek egy **bool** típusú kifejezésnek kell lennie) csak a függvény paramétereiből, literálokból és globális változókból álló (**bool**

---

<sup>1</sup>A **near** nem kulcsszó, csak a **raise** és a **near** együtt. A két szó között akárhány fehér szóköz állhat, vagyis a **raisenear** is helyes.

típusú) kifejezés állhat.

```
function foo(<args>)  
pre { <cond> }  
{  
    // ...  
}
```

Az utófeltétel megadása a **post** kulcsszó segítségével történik. A **post** paramétere egy változó lista, amelynek hossza a visszatérési értékek számával meg kell egyezzen – ezekkel az azonosítókkal lehet hivatkozni a visszatérési értékekre. A **post** blokkja két részből áll: egy értékadásból és egy **bool** típusú kifejezésből. A két részt egy **;** (pontosvessző) köti össze. Az értékadás jobb- és baloldalán is csak azonosítók szerepelhetnek. A baloldalon lévő azonosítók típusát a jobboldalból vezeti le a fordító. Jobboldalon csak a függvény paraméterei állhatnak. Ezzel a módszerrel lehet a paraméterek kezdeti értékét eltárolni. A feltétel az előző értékadás baloldalán szereplő azonosítókból, a **post** paramétereiből, literálokból és globális változókból álló (**bool** típusú) kifejezés.

```
function foo(<args>) : <types>  
post(<identlist>) { <assign> ; <cond> }  
{  
    // ...  
}
```

A következő példa az elő- és utófeltételek használatát mutatja be. A függvény két **u32** típusú számot oszt el. Az előfeltétele az, hogy az osztó

nem lehet **0u** (nulla), és az eredmény a két szám hányadosa.

```
function divide(u32 a, u32 b) : u32
pre { b!=0u }
post(r) { a0, b0 = a, b; r==a0/b0 }
{
    u32 n = 0u;
    while(a >= b)
    {
        a, n = a-b, n+1u;
    }
    return n;
}
```

### A.1.7. Elágazás

Az elágazás a C++ nyelv szintaxisához hasonló szintaxisú, de funkcionálisan inkább az Ada illetve az Eiffel nyelv elágazásaira hasonlít. Az A.6. ábrán egy többágú elágazás látható, ahol az **elif(<cond>) { }** összetevőből akárhány lehet (el is hagyható), valamint az **else { }** összetevő opcionális. A feltételek (<cond>) csak **bool** típusú kifejezések lehetnek.

```
if(<cond>)
{
}
elif(<cond>)
{
}
else
{
}
```

A.6. ábra. Az elágazás szintaxisa

Az elágazás szemantikája szerint pontosan egy ág fog végrehajtódni. A kiértékelés felülről lefelé halad, és az első **true** értékű kifejezéshez tartozó ág lefut, majd az elágazás végrehajtása véget ér. Ha az **else** ág hiányzik, akkor üres ágnak tekintendő, melynek feltétele **true** (nem fog utasításokat generálni a fordító hozzá).

### A.1.8. Ciklusok

A ScriptKernel nyelv négy különböző ciklus konstrukciót támogat. Ezek az **while**, a **do-while**, a **for** és **foreach**.

```
while(<cond>)  
{  
}
```

A.7. ábra. A **while** ciklus felépítése

A **while** ciklus a C illetve C++ nyelvekben megszokott előtesztelésű ciklus, melynek magja addig hajtódik végre, amíg a feltételben megadott **bool** típusú kifejezés **true**-ra értékelődik ki. A ciklus pontos felépítését az A.7. ábra szemlélteti.

```
do  
{  
}  
while(<cond>;
```

A.8. ábra. A **do-while** ciklus felépítése

A **do-while** ciklus a C illetve C++ nyelvekből ismert **do-while** ciklus alapján készült. Hátultesztelésű ciklus, a ciklus magja egyszer mindenképpen végrehajtásra kerül, és amíg a feltételként megadott **bool** típusú kifejezés **true**-ra értékelődik ki, addig újra és újra végrehajtódik a ciklus magja. Kilépés akkor, ha a feltétel **false**. Az általános felépítése az A.8. ábrán látható.

```
for(<assign_i>;<cond>;<assign_s>)
{
}
```

A.9. ábra. A **for** ciklus felépítése

Az A.9. ábrán látható **for** ciklus látszólag hasonlít a C és C++ nyelvekből ismert **for** ciklusra, azonban néhány fontos különbség van a kettő **for** ciklus között. A típusok szempontjából a legfontosabb különbsége a ciklus feltétele: a **cond** csak **bool** típusú kifejezés lehet, azonban megadása nem kötelező – ebben az esetben **true**-ként kezeli a fordító, így ha nem történik kivétel kiváltás vagy **return** utasítás, akkor egy végtelen ciklus jön létre (ha a **<cond>** hiányzik, akkor a **true** kifejezés nem kerül bele a kódba, mert mindig igazra értékelődik ki: feltétel nélküli ugrás kerül a lefordított programba). A C nyelv **for** ciklusának az inicializálása egy tetszőleges kifejezés lehet – a saját nyelvemben az **for** első komponense, az **<assign\_i>** egy (szimultán) értékadás. Ez a komponens is elhagyható. Ezt az értékadást a ciklusba való belépés előtt, egyszer hajtja végre. Hasonló a harmadik komponens, a **<assign\_s>** is, mely szintén egy értékadás lehet. Ha meg van adva, akkor ez az értékadás minden ciklusmag után egyszer lefut – ez a léptető értékadás.

```
//      1      2 3      4 5      6
foreach(<ident> in <lo> .. <hi> [inclusive/exclusive]
// 7      8      9
    [reverse] [while <cond_w>] [which <cond_f>])
{
}
```

A.10. ábra. A **foreach** ciklus intervallumokhoz

A nyelv legösszetettebb ciklusa a **foreach** ciklus, mely diszkrét intervallumokon, sorozatokon, valamint karakterláncokon képes iterálni. Az áttekinthetőbb leírás érdekében az A.11. ábrán hiányzik a felsorolásból

```

//      1      2  3
foreach(<ident> in <expr>
// 7      8      9
    [reverse] [while <cond_w>] [which <cond_f>])
{
}

```

A.11. ábra. A **foreach** ciklus sorozatokhoz és karakterláncokhoz

a 4, az 5 és a 6 – ennek oka, hogy szinkronban legyen az A.10. ábrával. A **foreach** ciklus első összetevője egy azonosító, melyek az iterált elemre lehet hivatkozni. Ezt az azonosítót az ábrák az 1-es számmal jelölik. A 2-es számú összetevő egy elválasztó: az **in** kulcsszó. Az A.10. ábrán látható 3, 4 és 5 számú összetevőkkel az intervallumot lehet leírni. Az intervallum alsó (**<lo>**) korlátja nem lehet nagyobb, mint a felső (**<hi>**) korlátja – ellentkező esetben egyetlen elemet sem fog a ciklus bejárni. A 4-es összetevő egy elválasztó, mely a **..** és a **to** kulcsszavak lehetnek. Az intervallum alsó- és felső korlátjának típusa meg kell egyezzen, és ez a típus lesz az 1-es összetevőben megadott változó típusa is (ez a típusok csak **i32**, **u32**, **i64** és **u64** típusok egyike lehet). Az A.11. ábra 3-as összetevője bármilyen sorozat (**seq<T>**), valamint karakterlánc (**string**) lehet. Ebben az esetben az 1-es összetevővel megadott változó típusa sorozat esetén **T**, **string** esetén **char**. A 6-os összetevő egy opcionális módosító, amely az intervallum határait állítja be. Az alapértelmezett az **inclusive**, melynek hatására az intervallum alulról és felülről is zárt intervallumot jelent, míg az **exclusive** esetben alulról zárt, felülről nyílt intervallumot eredményez. A 7-es összetevő az intervallum/sorozat/karakterlánc elemeinek bejárési sorrendjét változtatja meg. Ha meg van adva a **reverse** kulcsszó, akkor az elemeket fordított sorrendben járja be a ciklus (ez a módosító a bejárando intervallumot nem befolyásolja, vagyis pontosan ugyanazokat az elemeket fogja bejárni **reverse** módosítóval, mint anélkül). A 8-as összetevővel egy kilépési feltétel írható a ciklusba: amíg a megadott **bool** típusú ki-

fejezés **true**-ra értékelődik ki, addig tart a ciklusmag végrehajtása. A 9-es összetevővel a felsorolt elemeket lehet szűrni oly módon, hogyha a feltételként megadott **bool** típusú kifejezés **true**-ra értékelődik ki, akkor a ciklusmag lefut majd lép a következő elemre, ha **false**-ra, akkor a ciklusmag nem fut le, és lép a következő elemre.

## A.2. Típusok

Ebben a fejezetben az 1.2. fejezethez kapcsolódó fogalmak részletes magyarázata található.

### A.2.1. Alaptípusok

A nyelv alaptípusai között találhatóak szám típusok (egész, lebegőpontos), logikai típus, szöveggel kapcsolatos (karakterlánc, karakter), valamint speciális alaptípusok.

| Típus      | Álnév        | Szuffix   | Decimális | Oktális | Hexadecimális | Bináris  |
|------------|--------------|-----------|-----------|---------|---------------|----------|
| <b>i32</b> | <b>int</b>   | -         | 123       | 0o174   | 0xF9E4        | 0b1011   |
| <b>u32</b> | <b>uint</b>  | <b>u</b>  | 123u      | 0o174u  | 0xF9E4u       | 0b1011u  |
| <b>i64</b> | <b>long</b>  | <b>L</b>  | 123L      | 0o174L  | 0xF9E4l       | 0b1011L  |
| <b>u64</b> | <b>ulong</b> | <b>UL</b> | 123UL     | 0o174ul | 0xF9E4L       | 0b1011uL |

A.12. ábra. Egész típusok

Az A.12. ábrán az egész típusok láthatóak. Minden egész típusnak megvan a saját utótagja, amely a konstans típusát határozza meg. Az egész típusok nem csak decimális alakban írhatók le, rendelkezésre áll nyolcas (oktális), tizenhatos (hexadecimális) és kettes (bináris) számrendszer is. Az utótagot (**u**, **L**) és a számrendszer indikátort (**o**, **x**, **b**) lehet kis- és nagybetűvel is írni. A típus neve és az álneve ugyanazt a típust jelölik, az álnevet csak a lexikális elemző ismeri.

A lebegőpontos típusok az A.13. ábrán láthatóak. Ahogy az egész típusok esetén is, az utótag lehet kis- és nagybetű is (**f** vagy **F**), továbbá

| Típus      | Álnév         | Szuffix  | Példa  |
|------------|---------------|----------|--------|
| <b>f32</b> | <b>float</b>  | <b>f</b> | 10.14f |
| <b>f64</b> | <b>double</b> | -        | 10.14  |

A.13. ábra. Lebegőpontos típusok

az álnevet a lexikális elemző **f32** vagy **f64** típusként ismeri fel. A lebegőpontos számok az IEEE 754 [IEE13, JMN<sup>+</sup>10] szabvány szerinti számok.

A logikai típus neve **bool**, az értékkészlete pedig a **{false, true}**. Alapértelmezés szerint egy **bool** típusú változó a **false** értéket kapja. Elágazás-, ciklus-, elő- és utófeltételek csak **bool** típusú kifejezések lehetnek.

A **string** típus karakterláncokat tárol. A karakterlánc hosszának lekérdezése konstans idejű (a karakterlánc ismeri a saját hosszát). A karakterlánc indexelő operátorának segítségével a karakterek egyesével lekérdezhetőek és módosíthatóak. A C és C++ nyelvhöz hasonlóan szöveg literálok fordítási időben összevonásra kerülnek.

```
string s = "Első rész " "második rész";
i32 len = strlen(s); // 0(1)
char c = s[0u];
s[3u] = '0';
```

A nyelv tartalmaz két speciális típust is, a **handle** és a **dhandle**, melyek használata a nyelvben erősen korlátozott. Megengedett műveletek: értékadás, egyenlőség- és nemegyenlőségvizsgálat. A **handle** egy 32 bites érték, a **dhandle** pedig 64 bites érték. Ezeket a típusokat külső függvényről (például a virtuális gép futtatási környezetéhez hozzáadott C függvényről) kapott értékek tárolására lehet használni (például mutató).

## A.2.2. Rekordok

Minden (nem importált privát) rekordhoz tartozik három generált művelet, melyeket nem lehet módosítani. Ezek a rekord létrehozása, másolása és törlése. A másoló függvény teljes másolást alkalmaz („*deep copy*”).

```
record <identifier>
{
    <fields>
}

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

A.14. ábra. A rekord felépítése

A privát rekord mezőit csak az exportáló program fér hozzá. Ennek segítségével teljesen zárt típusok hozhatók létre. A felhasználó (importáló) program privát rekordot csak létrehozni, másolni és törölni tudja, de ezek a műveletek is csak a fordító program számára használhatók (a három alapl műveletet nem lehet a forráskódból meghívni). A privát rekordokat paraméterként is átadhatja egy másik függvénynek, de ha az ugyanabból a programból importált függvény, akkor a hívott függvény már hozzáférhet a privát rekord mezőihez.

Mivel a privát rekordot importáló programnak nincs tudomása a rekord méretéről és mezőiről, ezért ha megváltozik a privát rekord, akkor nem kell újrafordítani az importáló programot – míg a nem privát rekordot importáló (és fel is használó) programot újra kell fordítani, mert a virtuális gép nem fogja betölteni a régi rekord deklarációt importáló programot.

### A.2.3. Sorozatok

A fordító a sorozatok műveleteit fordítási időben generálja, így csak azok a műveletek kerülnek bele a program, amelyekre szükség van. A sorozat típus a fordító belső *template*-jének tekinthető. A sorozat műveletei:

**function loext(seq<T> s, T e) : seq<T>;** Létrehoz egy új sorozatot, amely eggyel hosszabb, mint az **s**, és az **e** elemmel kezdődik.

**function hiext(seq<T> s, T e) : seq<T>;** Létrehoz egy új sorozatot, amely eggyel hosszabb, mint az **s**, és az **e** elemmel végződik.

**function lorem(seq<T> s) : seq<T>;** Létrehoz egy új sorozatot, amely eggyel rövidebb, mint az **s**, és az **s** legelső elemét nem tartalmazza az eredmény. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function hirem(seq<T> s) : seq<T>;** Létrehoz egy új sorozatot, amely eggyel rövidebb, mint az **s**, és az **s** legutolsó elemét nem tartalmazza az eredmény. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function lopop(seq<T> s) : seq<T>, T;** Létrehoz egy új sorozatot, amely eggyel rövidebb, mint az **s**, és az **s** legelső elemét nem tartalmazza az eredmény. A második visszatérési érték a hiányzó legelső elem. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function hipop(seq<T> s) : seq<T>, T;** Létrehoz egy új sorozatot, amely eggyel rövidebb, mint az **s**, és az **s** legutolsó elemét nem tartalmazza az eredmény. A második visszatérési érték a hiányzó legutolsó elem. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function lov(seq<T> s) : T;** Az eredmény a sorozat legelső eleme. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function hiv(seq<T> s) : T;** Az eredmény a sorozat legutolsó eleme. Üres sorozat esetén **IndexOutOfRangeException** kivételt vált ki.

**function seqlen(seq<T> s) : u32;** Az eredmény a sorozat elemeinek száma (a sorozat hossza). A módszertanban ezt a függvényt **dom** függvénynek hívják.

**function subseq(seq<T> s, u32 offs, u32 cnt) : seq<T>;**  
Létrehoz egy új sorozatot, amely az **s** egy részsorozata lesz. Ha az intervallum nagyobb, mint az **s** sorozat értelmezési tartománya, akkor a két intervallumot elmetszi, és az eredmény intervallumot fogja használni a részsorozat kiválasztásakor. Az **offs** a kezdő index, a **cnt** a kiválasztandó elemek száma.

**function seqfill(T e, u32 n) : seq<T>;** Létrehoz egy új, **n** elemű sorozatot, melyben minden eleme az **e** lesz.

A sorozatok elemeihez az indexelő operátorral lehet hozzáférni (olvasásra és írásra is). Az indexelő operátor paramétere egy **u32** típusú kifejezés.

```
seq<bool> sb;  
sb = loext(sb, false);  
sb = hiext(sb, true);  
bool lo = sb[0u]; // false  
bool hi = sb[seqlen(sb)-1u]; // true
```

A **string** speciális sorozatnak tekinthető, mert alaptípusa a **char**, és akárhány karaktert tartalmazhat. Azonban a **string** típus nem azonos a **seq<char>** típussal.

#### A.2.4. A mutable

A rekord, a sorozat és a **string** alapértelmezett paraméter átadási módja a konstans referencia (a paraméterül átadott rekordot, sorozat-

tot és **string**-et referencia, tulajdonképpen cím szerint adja át, de az így átadott értékek nem módosíthatóak). Lehetőség van arra, hogy az átadott adatszerkezeteket a hívott függvény módosítsa (mezők vagy elemek megváltoztatása illetve felülírása).

Ahhoz, hogy a hívott függvény megváltoztathassa az átadott objektum tartalmát, azt típus elé írt **mutable** módosítóval kell jelezni. Fontos, hogy **mutable** nem vesz részt a túlterhelési mechanizmusban, azaz ezen keresztül nem lehet túlterhelni a függvényeket.

A **mutable** egyfajta fordított pszichológia a C++ nyelv **const** kulcsszavának használatára. Kezdő programozók esetén gyakori hiba az, hogy paraméter átadáskor nem írják ki a **const** kulcsszót, ezáltal főlegesen lekorlátozzák a függvény felhasználását. Ezzel szemben a **mutable** használata kikényszeríti azt, hogy explicit jelezzék a paraméternél azt, hogy ez megváltoztatható. A **mutable** kulcsszó elhagyása esetén pedig a konstans adatszerkezetet nem lehet megváltoztatni, így jóval több esetben fogja a kezdő programozó is a helyes paraméterátadási módot használni.

```
function foo(mutable string s)
pre { strlen(s)>0 }
{
    char c = s[0u]; // ok
    s[0u] = 'a'; // ok
}

function bar(string s)
pre { strlen(s)>0 }
{
    char c = s[0u]; // ok
    s[0u] = 'a'; // fordítási hiba
}
```

A fenti példa esetén egy **function foo(string s);** függvényt már nem lehetne létrehozni, mert a túlterhelési mechanizmusban a **mutable**

módosítót törli a szignatúrából, így az új **foo** már többszörös definíció lenne.

### A.3. `complex.sks`: Komplex számok típusa

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "Komplex számok típusa"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

export
{
    record Complex
    {
        double re;
        double im;
    }
}

export
{
    operator ^= left 10;
}

export
{
    function "~"(Complex c) : Complex;
    function "=="(Complex lhs, Complex rhs) : bool;
    function "!="(Complex lhs, Complex rhs) : bool;
    function "+"(Complex lhs, Complex rhs) : Complex;
    function "-"(Complex lhs, Complex rhs) : Complex;
    function "*" (Complex lhs, Complex rhs) : Complex;
    function "/"(Complex lhs, Complex rhs) : Complex;
}

function "~"(Complex c) : Complex
{
    Complex x;
    x.re, x.im = c.re, -c.im;
    return x;
}

function "=="(Complex lhs, Complex rhs) : bool
{
    return lhs.re==rhs.re &&& -lhs.im==rhs.im;
}

function "!="(Complex lhs, Complex rhs) : bool
{
    return lhs.re==rhs.re &&& lhs.im==rhs.im;
}

function "+"(Complex lhs, Complex rhs) : Complex
{
    Complex x;
    x.re, x.im = lhs.re + rhs.re, lhs.im + rhs.im;
    return x;
}

function "-"(Complex lhs, Complex rhs) : Complex
{

```

```

    Complex x;
    x.re, x.im = lhs.re - rhs.re, lhs.im - rhs.im;
    return x;
}

function "(Complex lhs, Complex rhs) : Complex
{
    Complex x;
    x.re, x.im = lhs.re*rhs.re - lhs.im*rhs.im, lhs.im*rhs.re + lhs.re*rhs.im;
    return x;
}

function "/"(Complex lhs, Complex rhs) : Complex
//pre { rhs.re!=0.0 && rhs.im!=0.0 }
{
    double len = rhs.re*rhs.re + rhs.im*rhs.im;
    Complex x;
    x.re, x.im = (lhs.re*rhs.re + lhs.im*rhs.im)/len, (lhs.im*rhs.re - lhs.re*rhs.im)/len;
    return x;
}

main
{
    // Complex number library.
}

```

### A.3.1. A fordító által generált import include

```

// Generated import include for "complex.sks".
// ScriptKernel Compiler v2000
// Date: 2013.06.13. 23:13:55

#once _complex_sks__ImportInclude

// Operators.
import
{
    operator ^= left 10;
}

// Types.
import
{
    record Complex
    {
        f64 re;
        f64 im;
    }
}

// Functions.
import
{
    function "(Complex lhs, Complex rhs) : Complex;
    function "+(Complex lhs, Complex rhs) : Complex;
    function "-(Complex lhs, Complex rhs) : Complex;
    function "/"(Complex lhs, Complex rhs) : Complex;
    function "==(Complex lhs, Complex rhs) : bool;
    function "^(Complex lhs, Complex rhs) : bool;
    function "~(Complex c) : Complex;
}

```

## A.4. 128.sks: A 128 bites előjel nélküli egész típus

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "128 bites előjel nélküli egész szám megvalósítása"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

export
{
    record UInt128 = private;

    operator ** left 15;
}

export
{
    function tos(UInt128 i) : string;
    function tohex(UInt128 i) : string;
    function tou128(string s) : UInt128;
    function tou128(u32 n) : UInt128;
    function tou128(u64 n) : UInt128;
    function Make(mutable UInt128 i, u64 n);
    function Make(mutable UInt128 i, u64 hi, u64 lo);
    function "+"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "+"(UInt128 lhs, u32 n) : UInt128;
    function "+"(UInt128 lhs, u64 n) : UInt128;
    function "-"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "*" (UInt128 lhs, UInt128 rhs) : UInt128;
    function "/" (UInt128 lhs, UInt128 rhs) : UInt128;
    function "&"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "|" (UInt128 lhs, UInt128 rhs) : UInt128;
    function "^"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "~"(UInt128 n) : UInt128;
    function "<<"(UInt128 n, u32 s) : UInt128;
    function ">>"(UInt128 n, u32 s) : UInt128;
    function "=="(UInt128 lhs, UInt128 rhs) : bool;
    function "!="(UInt128 lhs, UInt128 rhs) : bool;
    function "<"(UInt128 lhs, UInt128 rhs) : bool;
    function ">"(UInt128 lhs, UInt128 rhs) : bool;
    function "<="(UInt128 lhs, UInt128 rhs) : bool;
    function ">="(UInt128 lhs, UInt128 rhs) : bool;
    function "***"(UInt128 n, u32 f) : UInt128;
}

declare
{
    record UInt128
    {
        u64 high;
        u64 low;
    }

    operator += left 4;
}

declare
{
    function "+="(mutable UInt128 lhs, UInt128 rhs);
    function "+="(mutable UInt128 lhs, u32 n);
    function "+="(mutable UInt128 lhs, u64 n);

    function "+"(char ch, string s) : string;
```

```

    function "<="(char lhs, char rhs) : bool;
    function HexChar(u64 N) : char;
    function ToHex(u64 x) : string;
}

import
{
    exception DivisionByZeroException;
}

function "+"(char ch, string s) : string
{
    return tos(ch) + s;
}

function "<="(char lhs, char rhs) : bool
{
    return (lhs - rhs)<=0;
}

function HexChar(u64 N) : char
{
    i32 n = toi32(N);

    if(n<10)
    {
        return '0' + n;
    }

    return 'A' + (n - 10);
}

function ToHex(u64 x) : string
{
    string s;

    do
    {
        s, x = HexChar(x & 0xful) + s, x >> 4ul;
    }
    while(x!=0ul);

    return s;
}

function tos(UInt128 i) : string
{
    if(i.high==0ul)
    {
        return tos(i.low);
    }

    u64 a0, a1, a2, a3 = i.high>>32ul, i.high&0xffffffffful, i.low>>32ul, i.low&0xffffffffful;
    u64 d, r;
    string s;

    do
    {
        r = a0;

        d = r / 10ul;
        r = ((r - d * 10ul) << 32ul) + a1;
        a0 = d & 0xffffffffful;

        d = r / 10ul;
        r = ((r - d * 10ul) << 32ul) + a2;
        a1 = d & 0xffffffffful;
    }

```

```

        d = r / 10ul;
        r = ((r - d * 10ul) << 32ul) + a3;
        a2 = d & 0xfffffffful;

        d = r / 10ul;
        r = r - d * 10ul;
        a3 = d;

        s = tos('0' + toi32(r)) + s;
    }
    while(a0!=0ul || a1!=0ul || a2!=0ul || a3!=0ul);

    return s;
}

function tohex(UInt128 i) : string
{
    if(i.high==0ul)
    {
        return ToHex(i.low);
    }
    else
    {
        string s = ToHex(i.low);
        u32 len = strlenu(s);

        if(len<16u)
        {
            s = substr("0000000000000000", 0u, 16u - len) + s;
        }

        return ToHex(i.high) + s;
    }
}

function tou128(string s) : UInt128
{
    UInt128 res;

    foreach(ch in s while '0'<=ch && ch<='9')
    {
        res += tou32(ch - '0');
    }

    return res;
}

function tou128(u32 n) : UInt128
{
    UInt128 res;
    res.low = tou64(n);
    return res;
}

function tou128(u64 n) : UInt128
{
    UInt128 res;
    res.low = n;
    return res;
}

function Make(mutable UInt128 i, u64 n)
{
    i.high, i.low = 0ul, n;
}

```

```

function Make(mutable UInt128 i, u64 hi, u64 lo)
{
    i.high, i.low = hi, lo;
}

function "=="(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high==rhs.high && lhs.low==rhs.low;
}

function "!="(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high!=rhs.high ||| lhs.low!=rhs.low;
}

function "<"(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high<rhs.high ||| (lhs.high==rhs.high && lhs.low<rhs.low);
}

function ">"(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high>rhs.high ||| (lhs.high==rhs.high && lhs.low>rhs.low);
}

function "<="(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high<rhs.high ||| (lhs.high==rhs.high && lhs.low<=rhs.low);
}

function ">="(UInt128 lhs, UInt128 rhs) : bool
{
    return lhs.high>rhs.high ||| (lhs.high==rhs.high && lhs.low>=rhs.low);
}

function "&"(UInt128 lhs, UInt128 rhs) : UInt128
{
    UInt128 res;
    res.high, res.low = lhs.high & rhs.high, lhs.low & rhs.low;
    return res;
}

function "|"(UInt128 lhs, UInt128 rhs) : UInt128
{
    UInt128 res;
    res.high, res.low = lhs.high | rhs.high, lhs.low | rhs.low;
    return res;
}

function "^"(UInt128 lhs, UInt128 rhs) : UInt128
{
    UInt128 res;
    res.high, res.low = lhs.high ^ rhs.high, lhs.low ^ rhs.low;
    return res;
}

function "~"(UInt128 n) : UInt128
{
    UInt128 res;
    res.high, res.low = ~n.high, ~n.low;
    return res;
}

function "+"(UInt128 lhs, UInt128 rhs) : UInt128
{
    UInt128 sum;

```

```

        sum.high, sum.low = lhs.high + rhs.high, lhs.low + rhs.low;
        if(sum.low < lhs.low)
        {
            sum.high = sum.high + 1ul;
        }

        return sum;
    }

function "+"(UInt128 lhs, u32 n) : UInt128
{
    UInt128 sum;

    sum.high, sum.low = lhs.high, lhs.low + tou64(n);
    if(sum.low < lhs.low)
    {
        sum.high = sum.high + 1ul;
    }

    return sum;
}

function "+"(UInt128 lhs, u64 n) : UInt128
{
    UInt128 sum;

    sum.high, sum.low = lhs.high, lhs.low + n;
    if(sum.low < lhs.low)
    {
        sum.high = sum.high + 1ul;
    }

    return sum;
}

function "+="(mutable UInt128 lhs, UInt128 rhs)
{
    u64 low = lhs.low;
    lhs.high, lhs.low = lhs.high + rhs.high, lhs.low + rhs.low;
    if(lhs.low < low)
    {
        lhs.high = lhs.high + 1ul;
    }
}

function "+="(mutable UInt128 lhs, u32 n)
{
    u64 low = lhs.low;
    lhs.high, lhs.low = lhs.high, lhs.low + tou64(n);
    if(lhs.low < low)
    {
        lhs.high = lhs.high + 1ul;
    }
}

function "+="(mutable UInt128 lhs, u64 n)
{
    u64 low = lhs.low;
    lhs.high, lhs.low = lhs.high, lhs.low + n;
    if(lhs.low < low)
    {
        lhs.high = lhs.high + 1ul;
    }
}

function "-"(UInt128 lhs, UInt128 rhs) : UInt128
{

```

```

    UInt128 diff;

    diff.high, diff.low = lhs.high - rhs.high, lhs.low - rhs.low;
    if(diff.low > lhs.low)
    {
        diff.high = diff.high - 1u;
    }

    return diff;
}

function "<<"(UInt128 n, u32 s) : UInt128
{
    UInt128 res;

    if(s >= 128u)
    {
        return res;
    }
    elif(s >= 64u)
    {
        res.high = n.low << tou64(s - 64u);
    }
    else // (s < 64u)
    {
        u64 s_ul = tou64(s);
        res.high, res.low = (n.high << s_ul) | (n.low >> tou64(64u - s)), (n.low << s_ul);
    }

    return res;
}

function ">>"(UInt128 n, u32 s) : UInt128
{
    UInt128 res;

    if(s >= 128u)
    {
        return res;
    }
    elif(s >= 64u)
    {
        res.low = n.high >> tou64(s - 64u);
    }
    else // (s < 64u)
    {
        u64 s_ul = tou64(s);
        res.high, res.low = (n.high >> s_ul), (n.high << tou64(64u - s)) | (n.low >> s_ul);
    }

    return res;
}

function "*" (UInt128 lhs, UInt128 rhs) : UInt128
{
    UInt128 res;
    UInt128 temp1 = lhs;
    UInt128 temp2 = rhs;

    while(temp1.high != 0u || temp1.low != 0u)
    {
        if((temp1.low & 0x1u) != 0u)
        {
            res = res + temp2;
        }

        temp1, temp2 = temp1 >> 1u, temp2 << 1u;
    }
}

```

```

    }

    return res;
}

function "/"(UInt128 lhs, UInt128 rhs) : UInt128
{
    u32    sh_cnt;
    UInt128 answer;
    UInt128 divisor = rhs;
    UInt128 dividend = lhs;
    UInt128 one;

    if(rhs.high==0ul && rhs.low==0ul)
    {
        raise(DivisionByZeroException);
    }

    one.low = 1ul;

    while(divisor < dividend)
    {
        divisor, sh_cnt = divisor << 1u, sh_cnt + 1u;
    }

    foreach(sh in 0u .. sh_cnt reverse)
    {
        if(divisor <= dividend)
        {
            answer, dividend = answer | (one << sh), dividend - divisor;
        }

        divisor = divisor >> 1u;
    }

    return answer;
}

function "*" (UInt128 n, u32 f) : UInt128
{
    UInt128 res;
    res.low = 1ul;

    foreach(i in 1u .. f)
    {
        res = res * n;
    }

    return res;
}

main
{
    // This is a library.
}

```

### A.4.1. A fordító által generált import include

```

// Generated import include for "128.sks".
// ScriptKernel Compiler v2000
// Date: 2013.06.07. 16:57:04

#once _128_sks__ImportInclude

```

```

// Operators.
import
{
    operator ** left 15;
}

// Types.
import
{
    record UInt128 = private;
}

// Functions.
import
{
    function "!="(UInt128 lhs, UInt128 rhs) : bool;
    function "&"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "*" (UInt128 lhs, UInt128 rhs) : UInt128;
    function "**"(UInt128 n, u32 f) : UInt128;
    function "+"(UInt128 lhs, u32 n) : UInt128;
    function "+"(UInt128 lhs, u64 n) : UInt128;
    function "+"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "-"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "/"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "<"(UInt128 lhs, UInt128 rhs) : bool;
    function "<<"(UInt128 n, u32 s) : UInt128;
    function "<=" (UInt128 lhs, UInt128 rhs) : bool;
    function "==" (UInt128 lhs, UInt128 rhs) : bool;
    function ">" (UInt128 lhs, UInt128 rhs) : bool;
    function ">=" (UInt128 lhs, UInt128 rhs) : bool;
    function ">>"(UInt128 n, u32 s) : UInt128;
    function "Make"(mutable UInt128 i, u64 n);
    function "Make"(mutable UInt128 i, u64 hi, u64 lo);
    function "^"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "tohex"(UInt128 i) : string;
    function "tos"(UInt128 i) : string;
    function "tou128"(string s) : UInt128;
    function "tou128"(u32 n) : UInt128;
    function "tou128"(u64 n) : UInt128;
    function "|"(UInt128 lhs, UInt128 rhs) : UInt128;
    function "~"(UInt128 n) : UInt128;
}

```

## A.5. test.sks: A 128 bites egész típus teszt-programja

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "Tesztprogram a 128 bites egész típoshoz"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

#include "128.import.inc"

export
{
    function pow(u64 a, u32 b) : string;
}

function pow(u64 a, u32 b) : string
{
    return tos(tou128(a)**b);
}

main
{
    UInt128 a, b;

    Make(a, 25000000000ul);
    write("a = " + tos(a) + "\n");
    Make(b, 10000000000000ul);
    write("b = " + tos(b) + "\n");

    UInt128 c = a * b;
    write("c = a * b = " + tos(c) + "\n");

    UInt128 d = c / a;
    write("d = c / a = " + tos(d) + "\n");

    UInt128 e = c / b;
    write("e = c / b = " + tos(e) + "\n");

    UInt128 base = tou128(1234u);
    string s_base = tos(base);
    foreach(x in 0u .. 12u)
    {
        write(s_base + " ** " + tos(x) + " = " + tos(base**x) + "\n");
    }
}
```

## A.6. gl.sks: OpenGL példaprogram

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "OpenGL példaprogram"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

import
{
    function glBegin(u32 mode);
    function glEnd();
}
```

```

    function glVertex3f(f32 x, f32 y, f32 z);
    function glColor3f(f32 r, f32 g, f32 b);
}

global
{
    u32 GL_TRIANGLES = 0x0004u;
    u32 GL_QUADS     = 0x0007u;
}

export
{
    record Mesh = private;
}

export
{
    record Vertex3f
    {
        float x;
        float y;
        float z;
    }

    record Color3f
    {
        float r;
        float g;
        float b;
    }
}

export
{
    record Triangle
    {
        bool visible;

        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Color3f  c0;
        Color3f  c1;
        Color3f  c2;
    }

    record Quad
    {
        bool visible;

        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Vertex3f v3;
        Color3f  c0;
        Color3f  c1;
        Color3f  c2;
        Color3f  c3;
    }
}

declare
{
    record Mesh
    {
        seq<Triangle> tris;
        seq<Quad>    quads;
    }
}

```

```

        float        radius;
    }
}

export
{
    function AddTriangle(mutable Mesh m, Triangle t);
    function AddTriangle(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Color3f c0, Color3f c1, Color3f c2);
    function AddQuad(mutable Mesh m, Quad q);
    function AddQuad(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Vertex3f v3, Color3f c0,
        Color3f c1, Color3f c2, Color3f c3);
    function BoundingRadius(Mesh m) : float;
    function Render(Mesh m);
    function Render(Triangle tri);
    function Render(Quad quad);
}

function UpdateMesh(mutable Mesh m, Vertex3f v)
{
    float dist = sqrt(v.x*v.x + v.y*v.y + v.z*v.z);

    if(m.radius < dist)
    {
        m.radius = dist;
    }
}

function AddTriangle(mutable Mesh m, Triangle t)
{
    m.tris = hiext(m.tris, t);
    UpdateMesh(m, t.v0);
    UpdateMesh(m, t.v1);
    UpdateMesh(m, t.v2);
}

function AddTriangle(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Color3f c0, Color3f c1, Color3f c2)
{
    Triangle t;
    t.visible = true;
    t.v0, t.v1, t.v2 = v0, v1, v2;
    t.c0, t.c1, t.c2 = c0, c1, c2;
    m.tris = hiext(m.tris, t);
    UpdateMesh(m, v0);
    UpdateMesh(m, v1);
    UpdateMesh(m, v2);
}

function AddQuad(mutable Mesh m, Quad q)
{
    m.quads = hiext(m.quads, q);
    UpdateMesh(m, q.v0);
    UpdateMesh(m, q.v1);
    UpdateMesh(m, q.v2);
    UpdateMesh(m, q.v3);
}

function AddQuad(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Vertex3f v3,
    Color3f c0, Color3f c1, Color3f c2, Color3f c3)
{
    Quad q;
    q.visible = true;
    q.v0, q.v1, q.v2, q.v3 = v0, v1, v2, v3;
    q.c0, q.c1, q.c2, q.c3 = c0, c1, c2, c3;
    m.quads = hiext(m.quads, q);
    UpdateMesh(m, v0);
    UpdateMesh(m, v1);
    UpdateMesh(m, v2);
    UpdateMesh(m, v3);
}

```

```

        UpdateMesh(m, v3);
    }

function BoundingRadius(Mesh m) : float
{
    return m.radius;
}

function Render(Mesh m)
{
    glBegin(GL_TRIANGLES);

    foreach(tri in m.tris which tri.visible)
    {
        glColor3f(tri.c0.r, tri.c0.g, tri.c0.b);
        glVertex3f(tri.v0.x, tri.v0.y, tri.v0.z);

        glColor3f(tri.c1.r, tri.c1.g, tri.c1.b);
        glVertex3f(tri.v1.x, tri.v1.y, tri.v1.z);

        glColor3f(tri.c2.r, tri.c2.g, tri.c2.b);
        glVertex3f(tri.v2.x, tri.v2.y, tri.v2.z);
    }

    glEnd();

    glBegin(GL_QUADS);

    foreach(quad in m.quads which quad.visible)
    {
        glColor3f(quad.c0.r, quad.c0.g, quad.c0.b);
        glVertex3f(quad.v0.x, quad.v0.y, quad.v0.z);

        glColor3f(quad.c1.r, quad.c1.g, quad.c1.b);
        glVertex3f(quad.v1.x, quad.v1.y, quad.v1.z);

        glColor3f(quad.c2.r, quad.c2.g, quad.c2.b);
        glVertex3f(quad.v2.x, quad.v2.y, quad.v2.z);

        glColor3f(quad.c3.r, quad.c3.g, quad.c3.b);
        glVertex3f(quad.v3.x, quad.v3.y, quad.v3.z);
    }

    glEnd();
}

function Render(Triangle tri)
{
    if(!tri.visible) { return; }

    glBegin(GL_TRIANGLES);

    glColor3f(tri.c0.r, tri.c0.g, tri.c0.b);
    glVertex3f(tri.v0.x, tri.v0.y, tri.v0.z);

    glColor3f(tri.c1.r, tri.c1.g, tri.c1.b);
    glVertex3f(tri.v1.x, tri.v1.y, tri.v1.z);

    glColor3f(tri.c2.r, tri.c2.g, tri.c2.b);
    glVertex3f(tri.v2.x, tri.v2.y, tri.v2.z);

    glEnd();
}

function Render(Quad quad)
{
    if(!quad.visible) { return; }

```

```

    glBegin(GL_QUADS);

    glColor3f(quad.c0.r, quad.c0.g, quad.c0.b);
    glVertex3f(quad.v0.x, quad.v0.y, quad.v0.z);

    glColor3f(quad.c1.r, quad.c1.g, quad.c1.b);
    glVertex3f(quad.v1.x, quad.v1.y, quad.v1.z);

    glColor3f(quad.c2.r, quad.c2.g, quad.c2.b);
    glVertex3f(quad.v2.x, quad.v2.y, quad.v2.z);

    glColor3f(quad.c3.r, quad.c3.g, quad.c3.b);
    glVertex3f(quad.v3.x, quad.v3.y, quad.v3.z);

    glEnd();
}

main
{
    // This is a library.
}

```

### A.6.1. A fordító által generált import include

```

// Generated import include for "gl.sks".
// ScriptKernel Compiler v2000
// Date: 2013.06.14. 15:35:10

#once _gl_sks__ImportInclude

// Types.
import
{
    record Mesh = private;
}
import
{
    record Vertex3f
    {
        f32 x;
        f32 y;
        f32 z;
    }
}
import
{
    record Color3f
    {
        f32 r;
        f32 g;
        f32 b;
    }
}
import
{
    record Triangle
    {
        bool visible;
        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Color3f c0;
        Color3f c1;
    }
}

```

```

        Color3f c2;
    }
}
import
{
    record Quad
    {
        bool visible;
        Vertex3f v0;
        Vertex3f v1;
        Vertex3f v2;
        Vertex3f v3;
        Color3f c0;
        Color3f c1;
        Color3f c2;
        Color3f c3;
    }
}

// Functions.
import
{
    function "AddQuad"(mutable Mesh m, Quad q);
    function "AddQuad"(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Vertex3f v3,
        Color3f c0, Color3f c1, Color3f c2, Color3f c3);
    function "AddTriangle"(mutable Mesh m, Triangle t);
    function "AddTriangle"(mutable Mesh m, Vertex3f v0, Vertex3f v1, Vertex3f v2, Color3f c0, Color3f c1, Color3f c2);
    function "BoundingRadius"(Mesh m) : f32;
    function "Render"(Mesh m);
    function "Render"(Triangle tri);
    function "Render"(Quad quad);
}

```

## A.7. gltest.sks: Az OpenGL példaprogram használata

```

#version 2.0
#meta:author "Barath, Aron"
#meta:title "OpenGL tesztprogram"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

#include "gl.import.inc"

export
{
    function Render();
}

global
{
    Mesh mesh;
}

function V(f32 x, f32 y, f32 z) : Vertex3f
{
    Vertex3f v;
    v.x, v.y, v.z = x, y, z;
    return v;
}

```

```

function White() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 1.f, 1.f, 1.f;
    return c;
}

function Black() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 0.f, 0.f, 0.f;
    return c;
}

function Grey() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 0.5f, 0.5f, 0.5f;
    return c;
}

function Red() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 1.f, 0.f, 0.f;
    return c;
}

function Green() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 0.f, 1.f, 0.f;
    return c;
}

function Blue() : Color3f
{
    Color3f c;
    c.r, c.g, c.b = 0.f, 0.f, 1.f;
    return c;
}

ctor
{
    Triangle tri;
    tri.v0, tri.c0 = V(1.f, 1.f, 1.f), White();
    tri.v1, tri.c1 = V(1.f, 2.f, 1.f), Grey();
    tri.v2, tri.c2 = V(2.f, 1.f, 2.f), Red();
    AddTriangle(mesh, tri);

    Quad quad;
    quad.v0, quad.c0 = V(0.f, 1.f, 2.f), Black();
    quad.v1, quad.c1 = V(1.f, 2.f, 3.f), Green();
    quad.v2, quad.c2 = V(2.f, 3.f, 0.f), Red();
    quad.v3, quad.c3 = V(1.f, 3.f, 1.f), Blue();
    AddQuad(mesh, quad);
}

main
{
    write("Bounding sphere radius: " + tos(BoundingRadius(mesh)) + "\n");
}

function Render()
{
    Render(mesh);
}

```

## A.8. petri\_demo.sks: A Petri-háló DSL használata

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "Using the PetriNet simulator"
#meta:*      "datetime" = __DATE__TIME__
#meta:*      "file" = __FILE__

#include "petri.import.inc"

main
{
    PetriNet pn;
    Init(pn);

    PNTrans t0 = pn += Trans() && "t0";
    PNTrans t1 = pn += Trans() && "t1";
    PNTrans t2 = pn += Trans() && "t2";

    PNPlace p1 = pn += Place() && "p1" && 0u;
    PNPlace p2 = pn += Place() && "p2" && 0u;

    pn += t0 -> p1 -> t1 -> p2 -> (2u, t2);
    /* // equivalent with:
    pn += t0 -> p1;
    pn += p1 -> t1;
    pn += t1 -> p2;
    pn += p2 -> (2u, t2);
    */

    write("M: " + Weights(pn, true) + "\n");

    pn <- t0;

    write("M: " + Weights(pn, true) + "\n");

    pn <- t0 + t1 + t1;

    write("M: " + Weights(pn, true) + "\n");

    pn <- t2;

    write("M: " + Weights(pn, true) + "\n");

    exception(PNBadRelationException)
    {
        write("Attempt to create a bad relation.\n");
    }

    exception(PNWeightErrorException)
    {
        write("Using a transition caused weight error.\n");
    }

    exception(PNBadPetriNetException)
    {
        write("PetriNet mismatch exception.\n");
    }
}
```

## A.9. csoki.sks: Csokiautomata implementálása Petri-háló DSL segítségével

```
#version 2.0
#meta:author "Barath, Aron"
#meta:title "Csokiautomata implementálása Petri-hálóval"
#meta:*      "datetime" = __DATETIME__
#meta:*      "file" = __FILE__

#include "petri.import.inc"

main
{
    PetriNet pn;
    Init(pn);

    PNPlace p0 = pn += Place() && "0" && 1u;
    PNPlace p5 = pn += Place() && "5" && 0u;
    PNPlace p10 = pn += Place() && "10" && 0u;
    PNPlace p15 = pn += Place() && "15" && 0u;
    PNPlace p20 = pn += Place() && "20" && 0u;

    PNTrans t0_5 = pn += Trans() && "+5";
    PNTrans t5_10 = pn += Trans() && "+5";
    PNTrans t0_10 = pn += Trans() && "+10";
    PNTrans t5_15 = pn += Trans() && "+10";
    PNTrans t10_15 = pn += Trans() && "+5";
    PNTrans t15_20 = pn += Trans() && "+5";
    PNTrans t10_20 = pn += Trans() && "+10";
    PNTrans t15_0 = pn += Trans() && "Kis";
    PNTrans t20_0 = pn += Trans() && "Nagy";

    pn += p0 -> t0_5 -> p5 -> t5_10 -> p10 -> t10_15 -> p15 -> t15_20 -> p20 -> t20_0 -> p0;
    pn += p0 -> t0_10 -> p10 -> t10_20 -> p20;
    pn += p5 -> t5_15 -> p15 -> t15_0 -> p0;

    bool run = true;

    while(run)
    {
        seq<PNTrans> ts = AvailableTrans(pn);

        //write("M: " + Weights(pn, true) + "\n");
        write("Available transitions:\n");

        foreach(i in 0u to seqlen(ts) exclusive)
        {
            write("\t" + TransName(ts[i]) + "\n");
        }

        string choice = read();

        if(strlen(choice)==0)
        {
            write("No input, exit...\n");
            run = false;
        }
        else
        {
            bool done = false;

            foreach(i in 0u to seqlen(ts) exclusive while !done)
            {
                if(strcmp(TransName(ts[i]), choice)==0)
```

```

        {
            pn <- ts[i];
            done = true;
        }
    }

    if(!done)
    {
        write("No such transition.\n");
    }
}

exception(PNBadRelationException)
{
    write("Attempt to create a bad relation.\n");
}

exception(PNWeightErrorException)
{
    write("Using a transition caused weight error.\n");
}

exception(PNBadPetriNetException)
{
    write("PetriNet mismatch exception.\n");
}
}

```